Basic concepts
Basic concepts of computer science.

Information & Information Processing

Data – Information – Knowledge

The content of the human mind can be classified into four categories:

- Data: symbols;
- Information: data that are processed to be useful; provides answers to "who", "what", "where", and "when" questions;
- Knowledge: understanding of data and information; answers "how" questions;
- Wisdom: evaluated understanding.

Data

Data consist of raw facts and figures - it does not have any meaning until it is processed and turned into something useful.

Data comes in many forms; the main ones are letters, numbers and symbols.

Data is a prerequisite to information. For example, the two data items below could represent some very important information:

DATA	INFORMATION
123424331911	Your winning Lottery ticket number
211192	Your Birthday

An organization sometimes has to decide on the nature and volume of data that is required for creating the necessary information.

Information

Information is the data that has been processed in such a way as to be meaningful to the person who receives it.

INFORMATION = DATA + CONTEXT + MEANING

Example

Consider the number19051890. It has no meaning or context. It is an instance of data.

If a context is given : it is a date (Vietnamese use French format ddmmyyyy). This allows us to register it as 19th May 1890. It still has no meaning and is therefore not information

Meaning : The birth date of Vietnamese President Ho Chi Minh.

This gives us all the elements required for it to be called 'information'

Knowledge

By knowledge we mean the human understanding of a subject matter that has been acquired through proper study and experience.

Knowledge is usually based on learning, thinking, and proper understanding of the problem area. It can be considered as the integration of human perceptive processes that helps them to draw meaningful conclusions.

Consider this scenario: A person puts his finger into very hot water.

Data gathered: The finger nerve sends pain data to the brain.

Processing: The brain considers the data and comes up with...

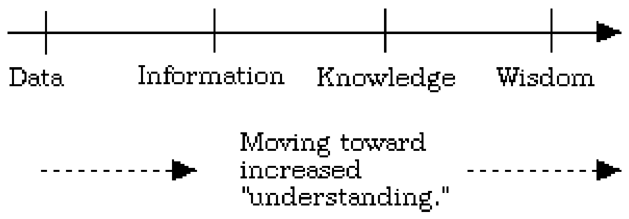
Information: The painful finger means it is not in a good place.

Action: The brain tells finger to remove itself from hot water.

Knowledge: Sticking the finger in hot water is a bad idea.

Knowledge is having an understanding of the "rules".

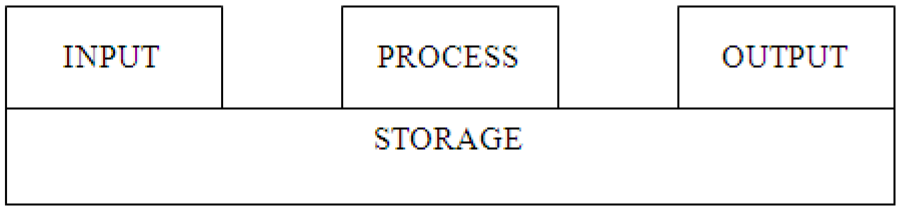
The terms Data, Information, Knowledge, and Wisdom are sometimes presented in a form that suggests a scale.



Data, Information, knowledge, wisdom along a scale

Information Processing

Information processing is the change (processing) of information in any manner detectable by an observer. Information processing may more specifically be defined in terms by Claude E. Shannon as the conversion of latent information into manifest.Input, process, output is a typical model for information processing. Each stage possibly requires data storage.



Model of information processing

Now that computer systems have become so powerful, some have been designed to make use of information in a knowledgeable way. The following definition is of information processing

The electronic capture, collection, storage, manipulation, transmission, retrieval, and presentation of information in the form of data, text, voice, or image and includes telecommunications and office automation functions.

History and Classification of Computers

History of Computers

Webster's Dictionary defines "computer" as any programmable electronic device that can store, retrieve, and process data.

Blaise Pascal invents the first commercial calculator, a hand powered adding machine

In 1946, ENIAC, based on John Von Neuman model completes.The first commercially successful computer is IBM 701.

A generation refers to the state of improvement in the development of a product. This term is also used in the different advancements of computer technology. With each generation, the circuitry has gotten smaller and more advanced than the previous generations before it. As a result of the miniaturization, the speed, power and memory of computers has proportionally increased. New discoveries are constantly being developed that affect the way we live, work and play. In terms of technological developments over time, computers have been broadly classed into five generations.

The First Generation - 1940-1956

The first computers used vacuum tubes for circuitry and magnetic drums for memory, and were often enormous, taking up entire rooms. They were very expensive to operate and in addition to using a great deal of electricity, they generated a lot of heat, which was often the cause of malfunctions. First generation computers relied on machine language to perform operations, and they could only solve one problem at a time. Input was based on punched cards and paper tape, and output was displayed on printouts.

The computers UNIVAC , ENIAC of the US and BESEM of the former Soviet Union are examples of first-generation computing devices.

The Second Generation - 1956-1963

Transistors replaced vacuum tubes and ushered in the second generation of computers. Computers becomed smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors. Second-generation computers still relied on punched cards for input and printouts for output. High-level programming languages were being developed, such as early versions of COBOL and FORTRAN.

The first computers of this generation were developed for the atomic energy industry.

The computers IBM-1070 of the US and MINSK of the former Soviet Union belonged to the second generation.

The Third Generation - 1964-1971: Integrated Circuits

The development of the integrated circuit was the hallmark of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers. Users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time. Typical computers of the third generation are IBM 360 (United States) and EC (former Soviet Union).

The Fourth Generation - 1971-Present: Microprocessors

The microprocessor brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip. What in the first generation filled an entire room could now fit in the palm of the hand. The Intel 4004 chip, developed in 1971, located all the components of the computer - from the central processing unit and memory to input/output controls - on a single chip.

In 1981 IBM introduced its first computer for the home user, and in 1984 Apple introduced the Macintosh. Microprocessors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use microprocessors.

As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUI (Graphic User Interface), the mouse and handheld devices.

The Fifth Generation - Present and Beyond: Artificial Intelligence

Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today. The use of parallel processing and superconductors is helping to make artificial intelligence a reality. Quantum computation and molecular and nanotechnology will radically change the face of computers in years to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization.

Classification of Computers

Computers are available in different shapes, sizes and weights, due to these different shapes and sizes they perform different sorts of jobs from one another.

- **Mainframe and Super Computers**

The biggest in size, the most expensive in price than any other is classified and known as super computer. It can process trillions of instructions in seconds. Governments specially use this type of computer for their different calculations and heavy jobs. This kind of computer is also helpful for forecasting weather reports worldwide.

Another giant in computers after the super computer is Mainframe, which can also process millions of instruction per second and capable of accessing billions of data. This computer is commonly used in big hospitals, airline reservations companies, and many other huge companies prefer mainframe because of its capability of retrieving data on a huge basis. This is normally too expensive and out of reach from a salary-based person who wants a computer for his home.

- **Minicomputers**

This computer offers less than mainframe in work and performance. These are the computers, which are mostly preferred by the small type of business personals, colleges, and so on.

- **Microcomputers**

These computers are lesser in cost than the computers given above and also, small in size; They can store a big amount of data and have a memory to meet the assignments of students and other necessary tasks of business people. There are many types of microcomputers: desktop, workstation, laptop, PDA , etc.

Computer Science and Relevant Sciences

In 1957 the German computer scientist Karl Steinbuch coined the word informatik by publishing a paper called Informatik: Automatische Informationsverarbeitung (i.e. "Informatics: automatic information processing"). The French term informatique was coined in 1962 by Philippe

Dreyfus together with various translations—informatics (English), informatica (Italian, Spanish, Portuguese), informatika (Russian) referring to the application of computers to store and process information.

The term was coined as a combination of "information" and "automation", to describe the science of automatic information processing.

Informatics is more oriented towards mathematics than computer science.

Computer Science

Computer Science is the study of computers, including both hardware and software design. Computer science is composed of many broad disciplines, for instance, artificial intelligence and software engineering.

Information Technology

Includes all matters concerned with the furtherance of computer science and technology and with the design, development, installation, and implementation of information systems and applications

Information and Communication Technology

ICT (information and communications technology - or technologies) is an umbrella term that includes any communication device or application, encompassing: radio, television, cellular phones, computer and network hardware and software, satellite systems and so on, as well as the various services and applications associated with them, such as videoconferencing and distance learning.

Data Representation in a Computer

Computer must not only be able to carry out computations, they must be able to do them quickly and efficiently. There are several data representations, typically for integers, real numbers, characters, and logical values.

Number Representation in Various Numeral Systems

A numeral system is a collection of symbols used to represent small numbers, together with a system of rules for representing larger numbers. Each numeral system uses a set of digits. The number of various unique digits, including zero, that a numeral system uses to represent numbers is called base or radix.

Base - b numeral system

b basic symbols (or digits) corresponding to natural numbers between 0 and b − 1 are used in the representation of numbers.

To generate the rest of the numerals, the position of the symbol in the figure is used. The symbol in the last position has its own value, and as it moves to the left its value is multiplied by b.

We write a number in the numeral system of base b by expressing it in the form
Equation:

$$N_{(b)} = a_n a_{n-1} a_{n-2} ... a_1 a_0 a_{-1} a_{-2} ... a_{-m}$$

N(b), with n+1 digit for integer and m digits for fractional part, represents the sum:

$$N_{(b)} = a_n.b^n + a_{n-1}.b^{n-1} + a_{n-2}.b^{n-2} + ... + a_1.b^1 + a_0.b^0 + a_{-1}.b^{-1} + a_{-2}.b^{-2} + ... + a_{-m}.b^{-m}$$

or

$$N_{(b)} = \sum_{i=-m}^{n} a_i.b^i$$

in the decimal system. Note that a_i is the i^{th} digit from the position of a_0

Decimal, Binary, Octal and Hexadecimal are common used numeral system. The decimal system has ten as its base. It is the most widely used numeral system, because humans have four fingers and a thumb on each hand, giving total of ten digit over both hand.

Switches, mimicked by their electronic successors built of vacuum tubes, have only two possible states: "open" and "closed". Substituting open=1 and closed=0 yields the entire set of binary digits. Modern computers use transistors that represent two states with either high or low voltages. Binary digits are arranged in groups to aid in processing, and to make the binary numbers shorter and more manageable for humans.Thus base 16 (hexadecimal) is commonly used as shorthand. Base 8 (octal) has also been used for this purpose.

Decimal System

Decimal notation is the writing of numbers in the base-ten numeral system, which uses various symbols (called digits) for no more than ten distinct values (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to represent any number, no matter how large. These digits are often used with a decimal separator which indicates the start of a fractional part, and with one of the sign symbols + (positive) or − (negative) in front of the numerals to indicate sign.

Decimal system is a place-value system. This means that the place or location where you put a numeral determines its corresponding numerical value. A two in the one's place means two times one or two. A two in the one-thousand's place means two times one thousand or two thousand.

The place values increase from right to left. The first place just before the decimal point is the one's place, the second place or next place to the left is the ten's place, the third place is the hundred's place, and so on.

The place-value of the place immediately to the left of the "decimal" point is one in all place-value number systems. The place-value of any place to the left of the one's place is a whole number computed from a product (multiplication) in which the base of the number system is repeated as a factor one less number of times than the position of the place.

For example, 5246 can be expressed like in the following expressions
Equation:

$$\begin{aligned}5246 \quad &= 5 \times 10^3 + 2 \times 10^2 + 4 \times 10^1 + 6 \times 10^0 \\ &= 5 \times 1000 + 2 \times 100 + 4 \times 10 + 6 \times 1\end{aligned}$$

The place-value of any place to the right of the decimal point is a fraction computed from a product in which the reciprocal of the base—or a fraction with one in the numerator and the base in the denominator—is repeated as a factor exactly as many times as the place is to the right of the decimal point.

For example
Equation:

$$\begin{aligned}254.68 &= 2 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 + 6 \times 10^{-1} + 8 \times 10^{-2} \\ &= 200 + 50 + 4 + \tfrac{6}{10} + \tfrac{8}{100}\end{aligned}$$

[missing_resource: graphics3.wmf]

Binary System

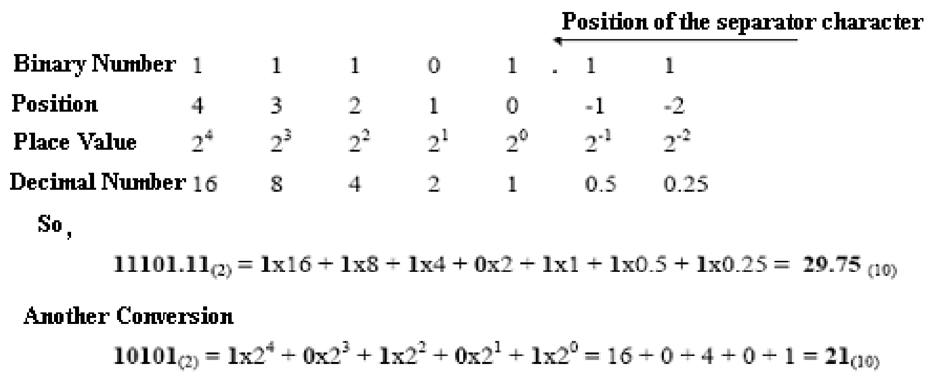
The binary number system is base 2 and therefore requires only two digits, 0 and 1. The binary system is useful for computer programmers, because it can be used to represent the digital on/off method in which computer chips and memory work.

A binary number can be represented by any sequence of bits (binary digits), which in turn may be represented by any mechanism capable of being in two mutually exclusive states.

Counting in binary is similar to counting in any other number system. Beginning with a single digit, counting proceeds through each symbol, in increasing order. Decimal counting uses the symbols 0 through 9, while binary only uses the symbols 0 and 1.

When the symbols for the first digit are exhausted, the next-higher digit (to the left) is incremented, and counting starts over at 0A single bit can represent one of two values, 0 or 1.Binary numbers are convertible to decimal numbers.

Here's an example of a binary number, $11101.11_{(2)}$, and its representation in the decimal notation



Position of the separator character

Binary Number	1	1	1	0	1	.	1	1
Position	4	3	2	1	0		-1	-2
Place Value	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}
Decimal Number	16	8	4	2	1		0.5	0.25

So ,

$11101.11_{(2)} = 1x16 + 1x8 + 1x4 + 0x2 + 1x1 + 1x0.5 + 1x0.25 = 29.75_{(10)}$

Another Conversion

$10101_{(2)} = 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 16 + 0 + 4 + 0 + 1 = 21_{(10)}$

Equation:

$$235.64_{(8)} = 2 \times 8^2 + 3 \times 8^1 + 5 \times 8^0 + 6 \times 8^{-1} + 4 \times 8^{-2} = 157.8125_{(10)}$$

Hexadecimal System

The hexadecimal system is base 16. Therefore, it requires 16 digits. The digits 0 through 9 are used, along with the letters A through F, which represent the decimal values 10 through 15. Here is an example of a hexadecimal number and its decimal equivalent:

Equation:

$$34F5C_{(16)} = 3 \times 16^4 + 4 \times 16^3 + 15 \times 16^2 + 5 \times 16^1 + 12 \times 16^0 = 216294_{(10)}$$

The hexadecimal system (often called the hex system) is useful in computer work because it is based on powers of 2. Each digit in the hex system is equivalent to a four-digit binary number. Table below shows some hex/decimal/binary equivalents.

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000

1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	10000
F0	240	11110000
FF	255	11111111

Octal System

Binary is also easily converted to the octal numeral system, since octal uses a radix of 8, which is a power of two (namely, 23, so it takes exactly three binary digits to represent an octal digit). The correspondence between octal and binary numerals is the same as for the first eight digits of hexadecimal in the table above. Binary 000 is equivalent to the octal digit 0, binary 111 is equivalent to octal 7, and so forth.

Converting from octal to binary proceeds in the same fashion as it does for hexadecimal:
Equation:

$$65_{(8)} = 110\,1012$$

Equation:

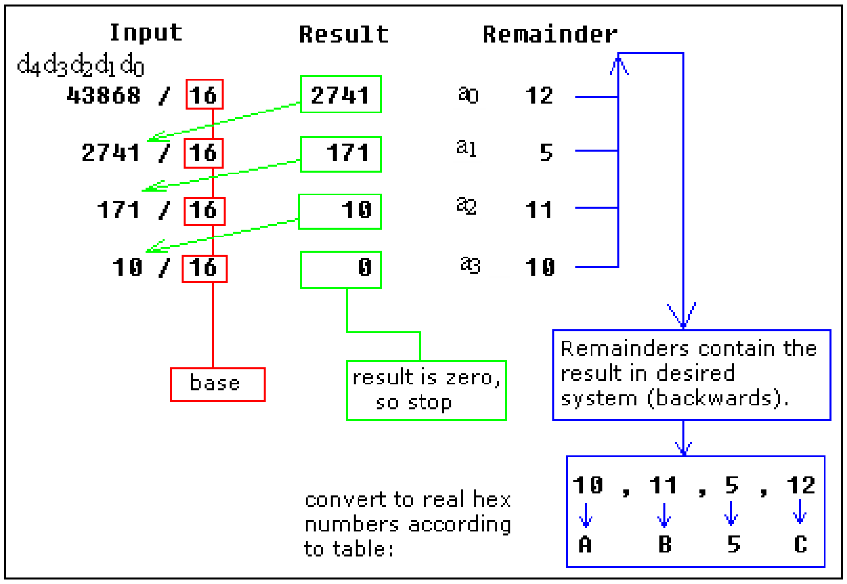
$$17_{(8)} = 001\ 1112$$

And from octal to decimal:
Equation:

$$235.64_{(8)} = 2 \times 8^2 + 3 \times 8^1 + 5 \times 8^0 + 6 \times 8^{-1} + 4 \times 8^{-2} = 157.8125_{(10)}$$

Converting from decimal to base–b

To convert a decimal fraction to another base, say base b, you split it into an integer and a fractional part. Then divide the integer by b repeatedly to get each digit as a remainder. Namely, with value of integer part = $d_{n-1}d_{n-2}...d_2d_1d_{0(10)}$, first divide value by b the remainder is the least significant digit a_0 . Divide the result by b, the remainder is a_1 .Continue this process until the result is zero, giving the most significant digit, a_{n-1} . Let's convert $43868_{(10)}$ to hexadecimal:



Converting from decimal to hexadecimal

After that, multiply the fractional part by b repeatedly to get each digit as an integer part. We will continue this process until we get a zero as our fractional part or until we recognize an infinite repeating pattern.

Now convert 0.625 to hexadecimal :

.

0.39625 * 16 = 0.625 -------------------------------------> 0

.625* 16 = 10 ---------------------------> A.

We get fractional part is zero.

In summary, the result of conversion 43868.39625$_{(10)}$ to hexadecimal is AB5C.0A

Data Representation in a Computer. Units of Information

Basic Principles

Data Representation refers to the methods used internally to represent information stored in a computer. Computers store lots of different types of information:

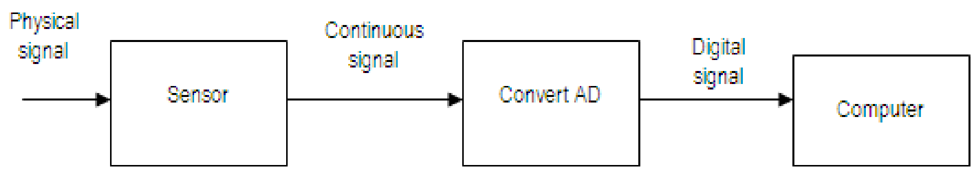
- numbers
- text
- graphics of many varieties (stills, video, animation)
- sound

At least, these all seem different to us. However, all types of information stored in a computer are stored internally in the same simple format: a sequence of 0's and 1's. How can a sequence of 0's and 1's represent things as diverse as your photograph, your favorite song, a recent movie, and your term paper?

- Numbers must be expressed in binary form following some specific standard.
- Character data are assigned a sequence of binary digits
- Other types of data, such as sounds, videos or other physical signals are converted to digital following the schema below

Digital signal

Continuous signalPhysical signalComputerConvert ADSensor



Process of converting from physical signal to digital signal

Depending on the nature of its internal representation, data items are divided into:

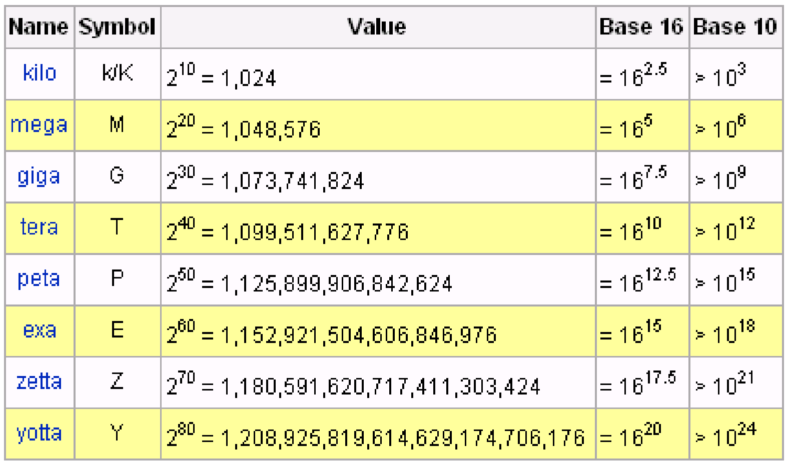
- Basic types (simple types or type primitives) : the standard scalar predefined types that one would expect to find ready for immediate use in any programming language
- Structured types(Higher level types) are then made up from such basic types or other existing higher level types.

Units of Information

The most basic unit of information in a digital computer is called a BIT, which is a contraction of Binary Digit. In the concrete sense, a bit is nothing more than a state of "on" or "off" (or "high" and "low") within a computer circuit. In 1964, the designers of the IBM System/360 mainframe computer established a convention of using groups of 8 bits as the basic unit of addressable computer storage. They called this collection of 8 bits a byte.

Computer words consist of two or more adjacent bytes that are sometimes addressed and almost always are manipulated collectively. The word size represents the data size that is handled most efficiently by a particular architecture. Words can be 16 bits, 32 bits, 64 bits, or any other size that makes sense within the context of a computer's organization.

Some other units of information are described in the following table :



Name	Symbol	Value	Base 16	Base 10
kilo	k/K	2^{10} = 1,024	$= 16^{2.5}$	$> 10^3$
mega	M	2^{20} = 1,048,576	$= 16^5$	$> 10^6$
giga	G	2^{30} = 1,073,741,824	$= 16^{7.5}$	$> 10^9$
tera	T	2^{40} = 1,099,511,627,776	$= 16^{10}$	$> 10^{12}$
peta	P	2^{50} = 1,125,899,906,842,624	$= 16^{12.5}$	$> 10^{15}$
exa	E	2^{60} = 1,152,921,504,606,846,976	$= 16^{15}$	$> 10^{18}$
zetta	Z	2^{70} = 1,180,591,620,717,411,303,424	$= 16^{17.5}$	$> 10^{21}$
yotta	Y	2^{80} = 1,208,925,819,614,629,174,706,176	$= 16^{20}$	$> 10^{24}$

Representation of Integers

An integer is a number with no fractional part; it can be positive, negative or zero. In ordinary usage, one uses a minus sign to designate a negative integer. However, a computer can only store information in bits, which can only have the values zero or one. We might expect, therefore, that the storage of negative integers in a computer might require some special technique - allocating one sign bit (often the most significant bit) to represent the sign: set that bit to 0 for a positive number, and set to 1 for a negative number.

Unsigned Integers

Unsigned integers are represented by a fixed number of bits (typically 8, 16, 32, and/or 64)

- With 8 bits, 0…255 (0016…FF16) can be represented;
- With 16 bits, 0…65535 (000016…FFFF16) can be represented;
- In general, an unsigned integer containing n bits can have a value between 0 and $2^n - 1$

If an operation on bytes has a result outside this range, it will cause an ‘overflow’

Signed Integers

The binary representation discussed above is a standard code for storing unsigned integer numbers. However, most computer applications use signed integers as well; i.e. the integers that may be either positive or negative.

In binary we can use one bit within a representation (usually the most significant or leading bit) to indicate either positive (0) or negative (1), and store the unsigned binary representation of the magnitude in the remaining bits.

However, for reasons of ease of design of circuits to do arithmetic on signed binary numbers (e.g. addition and subtraction), a more common representation scheme is used called two's complement. In this scheme, positive numbers are represented in binary, the same as for unsigned numbers. On the other hand, a negative number is represented by taking the binary representation of the magnitude:

- Complement the bits : Replace all the 1's with 0's, and all the 0's with 1's;
- Add one to the complemented number.

Example

```
+4210 =  001010102
and so
-4210   =  110101102
```

- Binary number with leading 0 is positive
- Binary number with leading 1 is negative

Example

Performing two's complement on the decimal 42 to get -42

Using a eight-bit representation

```
42= 00101010   Convert to binary

    11010101   Complement the bits

    11010101   Add 1 to the complement
  + 00000001
    --------
    11010110  Result is  -42 in two's complement
```

Arithmetic Operations on Integers

Addition and Subtraction of integers

Addition and subtraction of unsigned binary numbers

Binary Addition is much like normal everyday (decimal) addition, except that it carries on a value 2 instead of value 10.

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0, and carry 1 to the next more significant bit

Example

```
00011010 + 00001100 = 00100110
           1 1              carries
     0 0 0 1 1 0 1 0    =   26(base 10)
  +  0 0 0 0 1 1 0 0    =   12(base 10)
     ----------------
     0 0 1 0 0 1 1 0    =   38(base 10)
     11010001 + 00111110 = 100011010

     1 1           1          carries
     1 1 0 1 0 0 0 1     =    208 (base 10)
   + 0 1 0 0 1 0 0 1     =    73 (base 10)
     ---------------
   1 0 0 0 1 1 0 1 0     =    281 (base 10)
```

The result exceeds the magnitude which can be represented with 8 bits. This is an **overflow**.

Subtraction is executed by using two's complement

Addition and subtraction of signed binary numbers

Multiplication and Division of Integers

Binary Multiplication

Multiplication in the binary system works the same way as in the decimal system:

0 x 0 = 0

0 x 1 = 0

1 x 0 = 0

1 x 1 = 1, and no carry or borrow bits

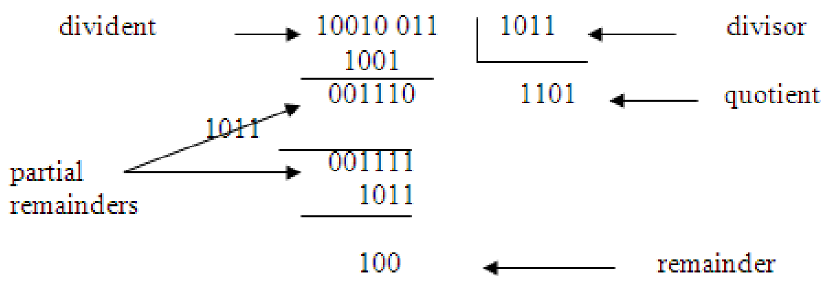
Example

```
00101001 × 00000110 = 11110110


      0  0  1  0  1  0  0  1       =    41(base 10)
×     0  0  0  0  0  1  1  0       =    6(base 10)
      ----------------------
         0  0  0  0  0  0  0
      0  1  0  1  0  0  1
```

```
   0  1  0  1  0  0  1
----------------------------
0  0  1  1  1  1  0  1  1  0        =    246(base 10)
```

```
00010111 × 00000011 = 01000101

         0  0  0  1  0  1  1  1        =    23(base 10)
×        0  0  0  0  0  0  1  1        =    3(base 10)
         ----------------------
               1  1  1  1  1                carries
            0  0  1  0  1  1  1
         0  0  1  0  1  1  1
   0  0  1  0  0  0  1  0  1           =    69(base 10)
```

Binary division follow the same rules as in decimal division.



Logical operations on Binary Numbers

Logical Operation with one or two bits

NOT : Changes the value of a single bit. If it is a "1", the result is "0"; if it is a "0", the result is "1".

AND: Compares 2 bits and if they are both "1", then the result is "1", otherwise, the result is "0".

OR : Compares 2 bits and if either or both bits are "1", then the result is "1", otherwise, the result is "0".

XOR : Compares 2 bits and if exactly one of them is "1" (i.e., if they are different values), then the result is "1"; otherwise (if the bits are the same), the result is "0".

Logical operators between two bits have the following truth table

x	y	x **AND** y	x **OR** y	x **XOR** y
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Logical Operation with one or two binary numbers

A logical (bitwise) operation operates on one or two bit patterns or binary numerals at the level of their individual bits.

Example

```
NOT 0111
  = 1000
```

AND operation

An AND operation takes two binary representations of equal length and performs the logical AND operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 AND the second bit is 1. Otherwise, the result is 0.

Example

```
     0101
```

```
AND  0011
   = 0001
```

OR operation

An OR operation takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits (the first of each; the second of each; and so on) and performing the logical OR operation on each pair of corresponding bits.

Example

```
      0101
   OR 0011
    = 0111
```

XOR Operation

An exclusive or operation takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits.

Example

```
     0101
 XOR 0011
   = 0110
```

Symbol Representation

Basic Principles

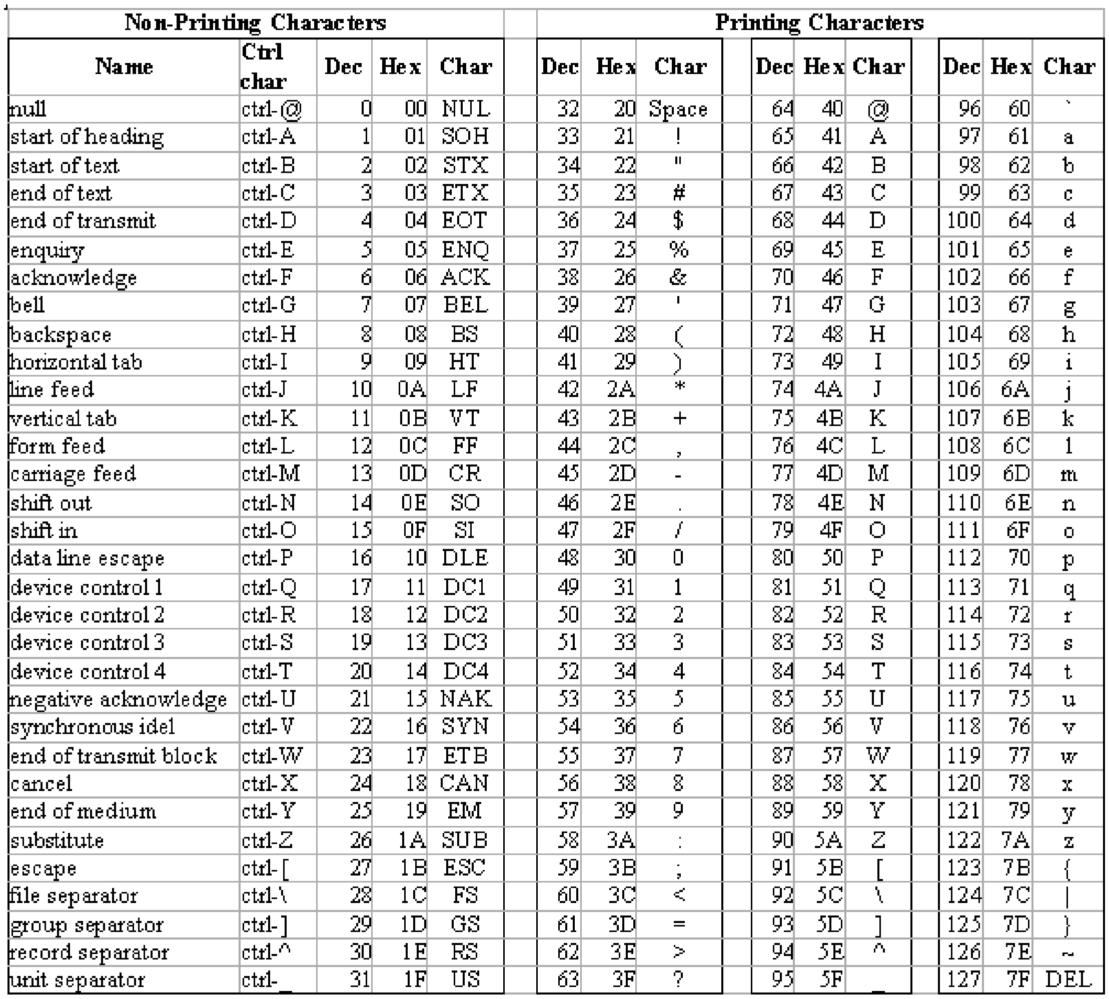
It is important to handle character data. Character data is not just alphabetic characters, but also numeric characters, punctuation, spaces, etc. They need to be represented in binary.

There aren't mathematical properties for character data, so assigning binary codes for characters is somewhat arbitrary.

ASCII Code Table

ASCII stands for American Standard Code for Information Interchange. The ASCII standard was developed in 1963, permitted machines from different manufacturers to exchange data.

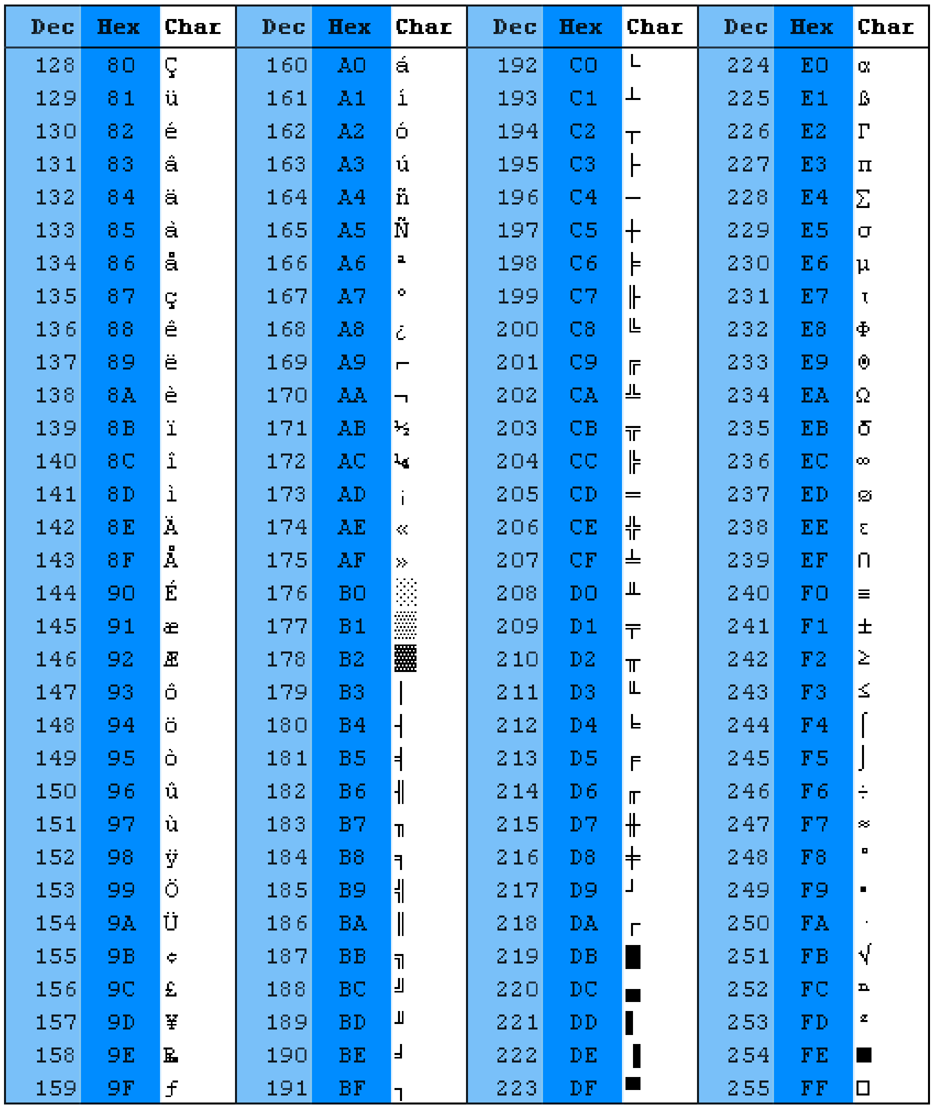
ASCII code table consists of 128 binary values (0 to 127), each associated with a character or command. The non-printing characters are used to control peripherals such as printer.



Non-Printing Characters					Printing Characters								
Name	Ctrl char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
null	ctrl-@	0	00	NUL	32	20	Space	64	40	@	96	60	`
start of heading	ctrl-A	1	01	SOH	33	21	!	65	41	A	97	61	a
start of text	ctrl-B	2	02	STX	34	22	"	66	42	B	98	62	b
end of text	ctrl-C	3	03	ETX	35	23	#	67	43	C	99	63	c
end of transmit	ctrl-D	4	04	EOT	36	24	$	68	44	D	100	64	d
enquiry	ctrl-E	5	05	ENQ	37	25	%	69	45	E	101	65	e
acknowledge	ctrl-F	6	06	ACK	38	26	&	70	46	F	102	66	f
bell	ctrl-G	7	07	BEL	39	27	'	71	47	G	103	67	g
backspace	ctrl-H	8	08	BS	40	28	(	72	48	H	104	68	h
horizontal tab	ctrl-I	9	09	HT	41	29	)	73	49	I	105	69	i
line feed	ctrl-J	10	0A	LF	42	2A	*	74	4A	J	106	6A	j
vertical tab	ctrl-K	11	0B	VT	43	2B	+	75	4B	K	107	6B	k
form feed	ctrl-L	12	0C	FF	44	2C	,	76	4C	L	108	6C	l
carriage feed	ctrl-M	13	0D	CR	45	2D	-	77	4D	M	109	6D	m
shift out	ctrl-N	14	0E	SO	46	2E	.	78	4E	N	110	6E	n
shift in	ctrl-O	15	0F	SI	47	2F	/	79	4F	O	111	6F	o
data line escape	ctrl-P	16	10	DLE	48	30	0	80	50	P	112	70	p
device control 1	ctrl-Q	17	11	DC1	49	31	1	81	51	Q	113	71	q
device control 2	ctrl-R	18	12	DC2	50	32	2	82	52	R	114	72	r
device control 3	ctrl-S	19	13	DC3	51	33	3	83	53	S	115	73	s
device control 4	ctrl-T	20	14	DC4	52	34	4	84	54	T	116	74	t
negative acknowledge	ctrl-U	21	15	NAK	53	35	5	85	55	U	117	75	u
synchronous idel	ctrl-V	22	16	SYN	54	36	6	86	56	V	118	76	v
end of transmit block	ctrl-W	23	17	ETB	55	37	7	87	57	W	119	77	w
cancel	ctrl-X	24	18	CAN	56	38	8	88	58	X	120	78	x
end of medium	ctrl-Y	25	19	EM	57	39	9	89	59	Y	121	79	y
substitute	ctrl-Z	26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
escape	ctrl-[	27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
file separator	ctrl-\	28	1C	FS	60	3C	<	92	5C	\	124	7C	|
group separator	ctrl-]	29	1D	GS	61	3D	=	93	5D	]	125	7D	}
record separator	ctrl-^	30	1E	RS	62	3E	>	94	5E	^	126	7E	~
unit separator	ctrl-_	31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

ASCII coding table

The extended ASCII character set also consists 128 128 characters representing additional special, mathematical, graphic and foreign characters.



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

The extended ASCII characters

Unicode Code Table

There are some problems with the ASCII code table. With ASCII character set, string datatypes allocated one byte per character. But logographic languages such as Chinese, Japanese, and Korean need far more than 256 characters for reasonable representation. Even Vietnamese, a language uses almost Latin letters, need 61 characters for representation. Where can we find numbers for our characters? is it a solution : 2 bytes per character?

Hundreds of different encoding systems were invented. But these encoding systems conflict with one another : two encodings can use the same number for two different characters, or use different numbers for the same character.

The Unicode standard was first published in 1991. With two bytes for each character, it can represent 216-1 different characters.

The Unicode standard has been adopted by such industry leaders as HP, IBM, Microsoft, Oracle, Sun, and many others. It is supported in many operating systems, all modern browsers, and many other products.

The obvious advantages of using Unicode are :

- To offer significant cost savings over the use of legacy character sets.
- To enable a single software product or a single website to be targeted across multiple platforms, languages and countries without re-engineering.
- To allow data to be transported through many different systems without corruption.

Representation of Real Numbers

Basic Principles

No human system of numeration can give a unique representation to real numbers. If you give the first few decimal places of a real number, you are giving an approximation to it.

Mathematicians may think of one approach : a real number x can be approximated by any number in the range from x - epsilon to x + epsilon. It is fixed-point representation. Fixed-point representations are unsatisfactory for most applications involving real numbers.

Scientists or engineers will probably use scientific notation: a number is expressed as the product of a mantissa and some power of ten.

A system of numeration for real numbers will typically store the same three data -- a sign, a mantissa, and an exponent -- into an allocated region of storage

The analogues of scientific notation in computer are described as floating-point representations.

In the decimal system, the decimal point indicates the start of negative powers of 10.
Equation:

$$12.34 = 1 * 10^1 + 2 * 10^0 + 3 * 10^{-1} + 4 * 10^{-2}$$

If we are using a system in base k (ie the radix is k), the ‘radix point’ serves the same function:
Equation:

$$\begin{aligned} 101.1012 &= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^2 \\ &= \ 4_{(10)} + 1_{(10)} + 0.5_{(10)} + 0.125_{(10)} \\ &= \ 5.625_{(10)} \end{aligned}$$

A floating point representation allows a large range of numbers to be represented in a relatively small number of digits by separating the digits used for precision from the digits used for range.

To avoid multiple representations of the same number floating point numbers are usually normalized so that there is only one nonzero digit to the left of the ‘radix’ point, called the leading digit.

A normalized (non-zero) floating-point number will be represented using
Equation:

$$(-1)^s d_0 \cdot d_1 d_2 ... d_{p-1} \times b^e$$

where

- s is the sign,
- $d_0 \cdot d_1 d_2 ... d_{p-1}$ - termed the significand - has p significant digits, each digit satisfies 0$\leq$ d_i <b
- e, $e_{\text{min}} \leq e \leq e_{\text{max}}$, is the exponent

- b is the base (or radix)

Example

If k = 10 (base 10) and p = 3, the number 0·1 is represented as 0.100

If k = 2 (base 2) and p = 24, the decimal number 0·1 cannot be represented exactly but is approximately $1 \cdot 10011001100110011001101 \times 2^{-4}$

Formally,

$(-1)^s d_0 \cdot d_1 d_2 ... d_{p-1}$ be represents the value
$(-1)^s (d_0 + d_1 b^{-1} + d_2 b^{-2} ... d_{-1} b^{(p-1)}) b^e$

In brief, a normalized representation of a real number consist of

- The range of the number : the number of digits in the exponent (i.e. by $e_{\max}$) and the base b to which it is raised
- The precision : the number of digits p in the significand and its base b

IEEE 754/85 Standard

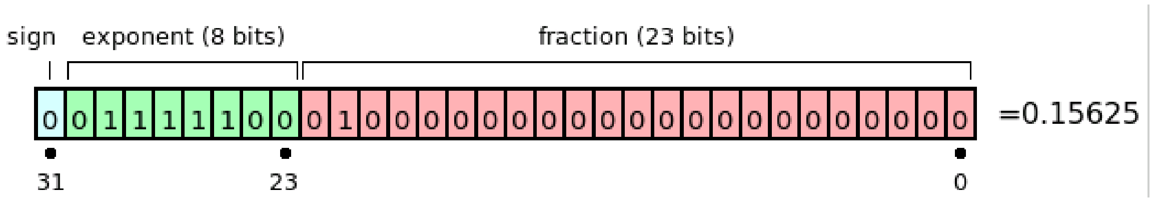
There are many ways to represent floating point numbers. In order to improve portability most computers use the IEEE 754 floating point standard.

There are two primary formats:

- 32 bit single precision.
- 64 bit double precision.

Single precision consists of:

- A single sign bit, 0 for positive and 1 for negative;
- An 8 bit base-2 (b = 2) excess-127 exponent, with $e_{\min}$ = –126 (stored as $127_{(10)} - 126_{(10)} = 1 = 00000001_{(2)}$) and $e_{\max}$ = 127 (stored as $127_{(10)} + 127_{(10)} = 254_{(10)} = 11111110_{(2)}$).
- a 23 bit base-2 (k=2) significand, with a hidden bit giving a precision of 24 bits (i.e. $1.d_1 d_2 ... d_{23}$)



Single precision memory format

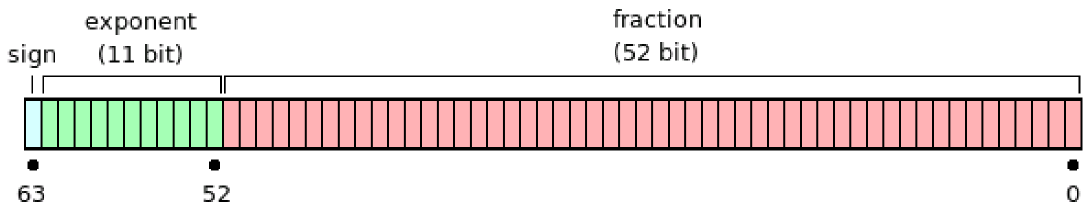
Notes

- Single precision has 24 bits precision, equivalent to about 7.2 decimal digits.
- The largest representable non-infinite number is almost $2 \times 2^{127} \cong 3.402823 \times 10^{38}$
- The smallest representable non-zero normalized number is $1 \times 2^{-127} \cong 1.17549 \times 10^{-38}$
- Denormalized numbers (eg 0.01×2^{-126}) can be represented.
- There are two zeros, $\pm$ 0.
- There are two infinities, $\pm\infty$.
- A NaN (not a number) is used for results from undefined operations

Double precision floating point standard requires a 64 bit word

- The first bit is the sign bit
- The next eleven bits are the exponent bits
- The final 52 bits are the fraction

Range of double numbers : [±2.225×10−308÷±1.7977×10308]



Double precision memory format

Computer Systems

A computer is an electronic device that performs calculations on data, presenting the results to humans or other computers in a variety of (hopefully useful) ways. The computer system includes not only the hardware, but also software that are necessary to make the computer function.

Computer hardware is the physical part of a computer, including the digital circuitry, as distinguished from the computer software that executes within the hardware.

Computer software is a general term used to describe a collection of computer programs, procedures and documentation that perform some task on a computer system

Computer Organization

General Model of a Computer

A computer performs basically five major operations or functions irrespective of their size and make.

1. Input: This is the process of entering data and programs in to the computer system. You should know that computer is an electronic machine like any other machine which takes as inputs raw data and performs some processing giving out processed data. Therefore, the input unit takes data from us to the computer in an organized manner for processing.

2. Storage: The process of saving data and instructions permanently is known as storage. Data has to be fed into the system before the actual processing starts. It is because the processing speed of Central Processing Unit (CPU) is so fast that the data has to be provided to CPU with the same speed. Therefore the data is first stored in the storage unit for faster access and processing. This storage unit or the primary storage of the computer system is designed to do the above functionality. It provides space for storing data and instructions.

The storage unit performs the following major functions:

- All data and instructions are stored here before and after processing.

- Intermediate results of processing are also stored here.

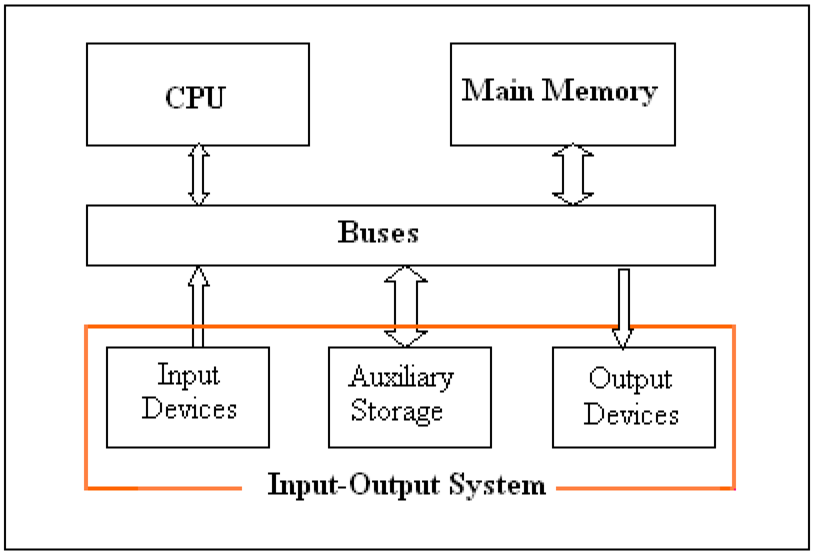
3. Processing: The task of performing operations like arithmetic and logical operations is called processing. The Central Processing Unit (CPU) takes data and instructions from the storage unit and makes all sorts of calculations based on the instructions given and the type of data provided. It is then sent back to the storage unit.

4. Output: This is the process of producing results from the data for getting useful information. Similarly the output produced by the computer after processing must also be kept somewhere inside the computer before being given to you in human readable form. Again the output is also stored inside the computer for further processing.

5. Control: The manner how instructions are executed and the above operations are performed. Controlling of all operations like input, processing and output are performed by control unit. It takes care of step by step processing of all operations in side the computer.

In order to carry out the operations mentioned above, the computer allocates the task between its various functional units. The computer system is divided into several units for its operation.

- CPU (central processing unit) : The place where decisions are made, computations are performed, and input/output requests are delegated
- Memory: stores information being processed by the CPU
- Input devices : allows people to supply information to computers
- Output devices : allows people to receive information from computers
- Buses : a bus is a subsystem that transfers data or power between computer components inside a computer.



General model of a computer

The basic function of a computer is program execution. When a program is running the executable binary file is copied from the disk drive into memory. The process of program execution is the retrieval of instructions and data from memory, and the execution of the various operations.The cycles with complex instruction sets typically utilize the following stages :

Fetch the instruction from main memory

The CPU presents the value of the program counter (PC) on the address bus. The CPU then fetches the instruction from main memory via the data bus into the Memory Data Register (MDR). The value from the MDR is then placed into the Current Instruction Register (CIR), a circuit that holds the instruction so that it can be decoded and executed.

Decode the instruction

The instruction decoder interprets and implements the instruction.

Fetch data from main memory

Read the effective address from main memory if the instruction has an indirect address. Fetch required data from main memory to be processed and placed into registers.

Execute the instruction

From the instruction register, the data forming the instruction is decoded by the control unit. It then passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the Arithmetic logic unit (ALU) to calculate the result and writing the result back to a register. A condition signal is sent back to the control unit by the ALU if it is involved.

Store results

The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition feedback from the ALU, the PC is either incremented to address the next instruction or updated to a different address where the next instruction will be fetched. The cycle is then repeated.

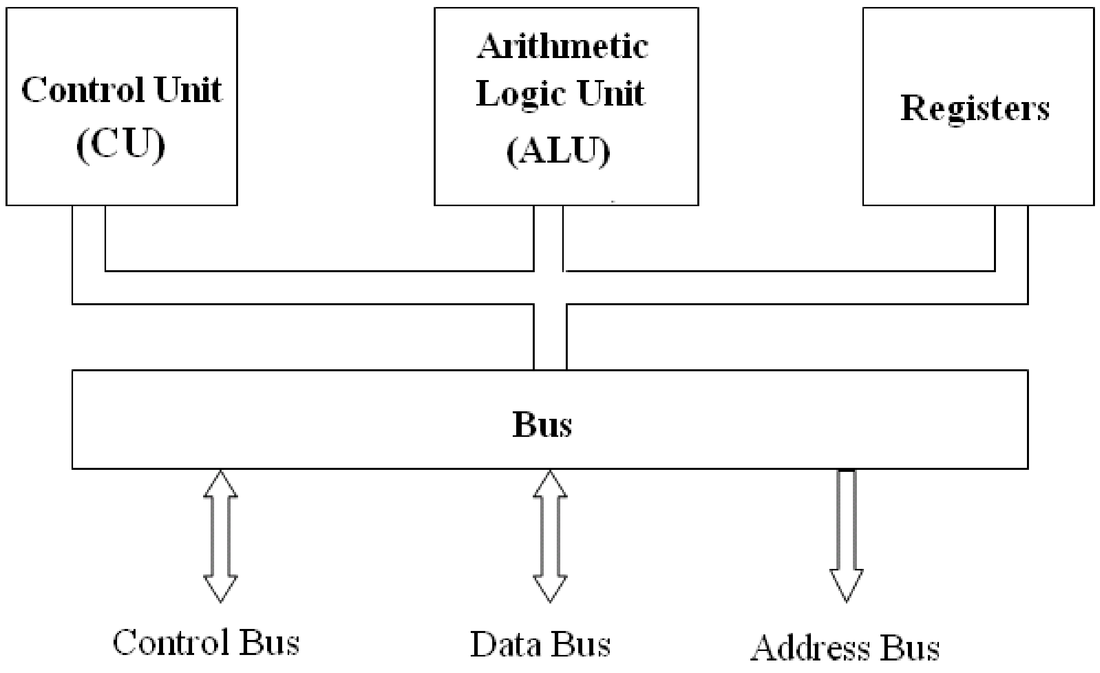
The Central Processing Unit (CPU)

You may call CPU as the brain of any computer system. It is the brain that takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.

CPU has four key parts

- Control Unit
- Arithmetic & Logic Unit
- Registers
- Clock

And, of course, wires that connect everything together.



Basic Model of the Central Processing Unit (CPU)

Arithmetic Logic Units (ALU)

The ALU, as its name implies, is that portion of the CPU hardware which performs the arithmetic and logical operations on the binary data .The ALU contains an Adder which is capable of combining the contents of two registers in accordance with the logic of binary arithmetic.

Control Unit

The control unit, which extracts instructions from memory and decodes and executes them, calling on the ALU when necessary.

Registers

Registers are temporary storage units within the CPU. Some registers, such as the program counter and instruction register, have dedicated uses. Other registers, such as the accumulator, are for more general purpose use.

Clock

A circuit in a processor that generates a regular sequence of electronic pulses used to synchronize operations of the processor's components. The time between pulses is the cycle time and the number of pulses per second is the clock rate (or frequency).

The execution times of instructions on a computer are usually measured by a number of clock cycles rather than seconds. The higher clock rate, the quicker speed of instruction processing. The clock rate for a Pentium 4 processor is about 2.0, 2.2 GHz or higher

Memory

Memory refer to computer components, devices and recording media that retain digital data used for computing for some interval of time. Computer memory includes internal and external memory.

Internal memory

The internal memory is accessible by a processor without the use of the computer input-output channels.It usually includes several types of storage, such as main storage, cache memory, and special registers, all of which can be directly accessed by the processor.

Cache memory : A buffer, smaller and faster than main storage, used to hold a copy of instructions and data in main storage that are likely to be needed next by the processor and that have been obtained automatically from main storage.

Main memory (Main Storage) : addressable storage from which instructions and other data may be loaded directly into registers for subsequent execution or processing.

Storage capacity of the main memory is the total amount of stored information that the memory can hold. It is expressed as a quantity of bits or bytes. Each address identifies a word of storage. So the capacity of the main memory depends on the number of bits allowed to address. For instance, a computer allows also 32-bit memory addresses; a byte-addressable 32-bit computer can address 2^{32} = 4,294,967,296 bytes of memory, or 4 gigabytes (GB). The capacity of the main memory is 4 GB.

The main memory consists of ROM and RAM.

- Random Access Memory (RAM): The primary storage is referred to as random access memory (RAM) because it is possible to randomly select and use any location of the memory directly store and retrieve data. It takes same time to any address of the memory as the first address. It is also called read/write memory. The storage of data and instructions inside the primary storage is temporary. It disappears from RAM as soon as the power to the computer is switched off.
- Read Only Memory (ROM): There is another memory in computer, which is called Read Only Memory (ROM). Again it is the ICs inside the PC that form the ROM. The storage of program and data in the ROM is permanent. The ROM stores some standard processing programs supplied by the manufacturers to operate the personal computer. The ROM can only be read by the CPU but it cannot be changed. The basic input/output program is stored in the ROM that examines and initializes various equipment attached to the PC when the switch is made ON.

External Memory

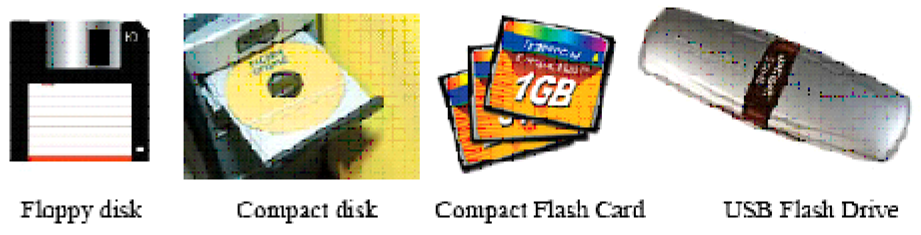
The external memory holds information too large for storage in main memory. Information on external memory can only be accessed by the CPU if it is first transferred to main memory. External memory is slow and virtually unlimited in capacity. It retains information when the computer is switched off and is used to keep a permanent copy of programs and data.

Hard Disk: is made of magnetic material. Magnetic disks used in computer are made on the same principle. It rotates with very high speed inside the computer drive. Data is stored on both the surface of the disk. Magnetic disks are most popular for direct access storage device. Each disk consists of a number of invisible concentric circles called tracks. Information is recorded on tracks of a disk surface in the form of tiny magnetic spots. The presence of a magnetic spot represents one bit and its absence represents zero bit. The information stored in a disk can be read many times without affecting the stored data. So the reading operation is non-destructive. But if you want to write a new data, then the existing data is erased from the disk and new data is recorded. The capacity of a hard disk is possibly 20 GB, 30 GB, 40 GB, 60 GB or more.

Floppy Disk: It is similar to magnetic disk discussed above. They are 5.25 inch or 3.5 inch in diameter. They come in single or double density and recorded on one or both surface of the diskette. The capacity of a 5.25-inch floppy is 1.2 mega bytes whereas for 3.5 inch floppy it is 1.44 mega bytes. The floppy is a low cost device particularly suitable for personal computer system.

Optical Disk:With every new application and software (includes sounds, images and videos) there is greater demand for memory capacity. It is the necessity to store large volume of data that has led to the development of optical disk storage medium. There are two commonly used categories of optical disks: CD with the approximate capacity of 700MB and DVD with the approximate capacity of 4.7GB

Memory Stick (Flash card, flash drive) a removable flash memory card format, with 128MB, 256 MB, 512 MB, 1 GB, 2 GB , 4 GB or more capacities

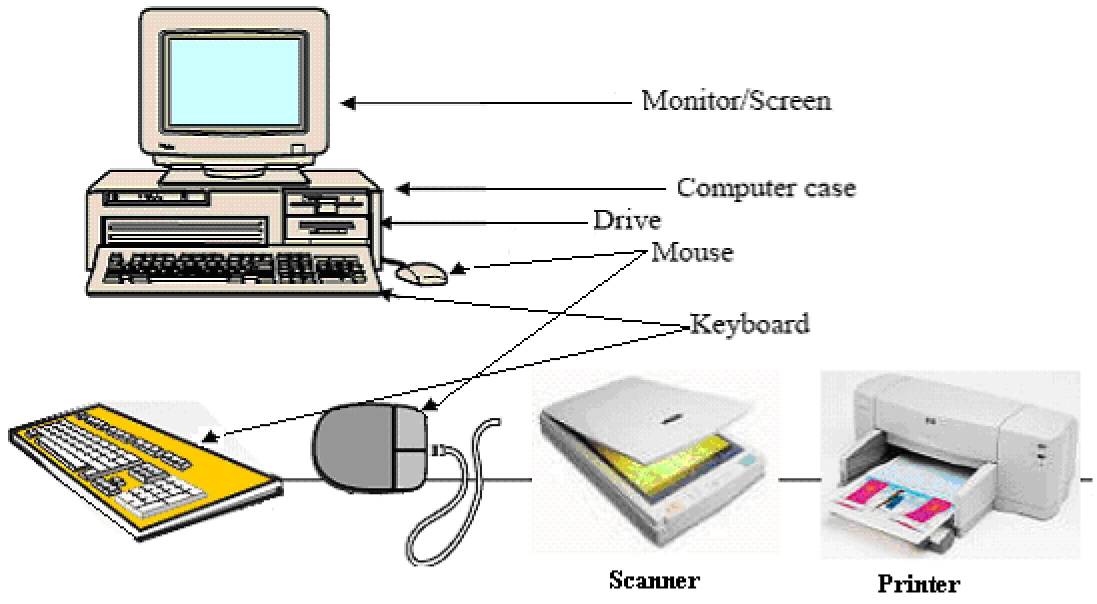


Some types of auxiliary memory

Input-Output Devices

A computer is only useful when it is able to communicate with the external environment. When you work with the computer you feed your data and instructions through some devices to the computer. These devices are called Input devices. Similarly the computer after processing, gives output through other devices called output devices.

Common input and output devices are: Speakers, Mouse, Scanner, Printer,Joystick, CD-ROM, Keyboard, Microphone, DVD, Floppy drive, Hard drive, Magnetic tape, and Monitor. Some devices are capable of both input and output.



Typical input- output devices

Input Devices

Input devices are necessary to convert our information or data in to a form which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing followings are the most useful input devices.

Keyboard: - This is the standard input device attached to all computers. The layout of keyboard is just like the traditional typewriter. It also contains some extra command keys and function keys. It contains a total of 101 to 104 keys. You must press correct combination of keys to input data. The computer can recognize the electrical signals corresponding to the correct key combination and processing is done accordingly.

Mouse: - Mouse is an input device that is used with your personal computer. It rolls on a small ball and has two or three buttons on the top.When you roll the mouse across a flat surface the screen censors the mouse in the direction of mouse movement. The cursor moves very fast with mouse giving you more freedom to work in any direction. It is easier and faster to move through a mouse.

Scanner: The keyboard can input only text through keys provided in it. If we want to input a picture the keyboard cannot do that. Scanner is an optical device that can input any graphical matter and display it back.

Output Devices

Monitor: The most popular input/output device is the monitor. A Keyboard is used to input data and Monitor is used to display the input data and to receive massages from the computer. A monitor has its own box which is separated from the main computer system and is connected to the computer by cable. It can be color or monochrome. It is controlled by an output device called a graphics card. Displayable area measured in dots per inch, dots are often referred to as pixels. Standard resolution is 640 by 480. Many cards support resolution of 1280 by 1024 or better. Number of colors supported varies from 16 to billions

Printer: It is an important output device which can be used to get a printed copy of the processed text or result on paper. There are different types of printers that are designed for different types of applications.

Buses

Bus is a subsystem that transfers data or power between computer components inside a computer or between computers. Bus can logically connect several peripherals over the same set of wires. Each bus defines its set of connectors to physically plug devices, cards or cables together. The buses are categorized depending on their tasks:

- The data bus transfers actual data.
- The address bus transfers information about where the data should go.
- The control bus carries signals that report the status of various devices.

Computer Software

Data and Algorithms

There are many steps involved in writing a computer program to solve a given problem. The steps go form problem formulation and specification, to design of the solution, to implementation, testing and documentation, and evaluation the solution.

Once we have a suitable mathematical model for our problem, we attempt to find a solution in term of that model. Our initial goal is to find a solution in the form of an algorithm. So what is an algorithm?

Algorithm is a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

How do you represent an algorithm? The most obvious representation: source code of a programming language. However, writing source code before you fully understand an algorithm often leads to difficult-to-find bugs. So, algorithms may be presented ...

1. In words

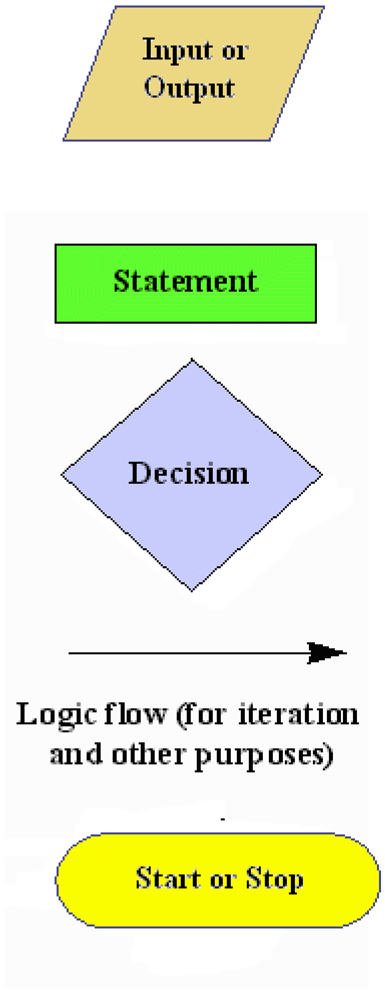
To present the algorithm in words we may describe the tasks step by step.

2.As a flowchart

A familiar technique for overcoming those bugs involves flowcharts.

A flowchart is a visual representation of an algorithm's control flow. That representation illustrates statements that need to execute, decisions that need to be made, logic flow (for iteration and other purposes), and terminals that indicate start and end points.

To present that visual representation, a flowchart uses various symbols, which Figure 3.5.shows.



Flowchart symbol for statements, decisions, logic flows, etc.

This review was essential because we will be using these building blocks quite often today.

3. In pseudocode

Pseudocode (derived from pseudo and code) is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of programming languages, but omits detailed subroutines, variable declarations or language-specific syntax. The programming language is augmented with natural language descriptions of the details, where convenient, or with compact mathematical notation.

Example

Present the algorithm of converting an integer from decimal to binary

a. By words

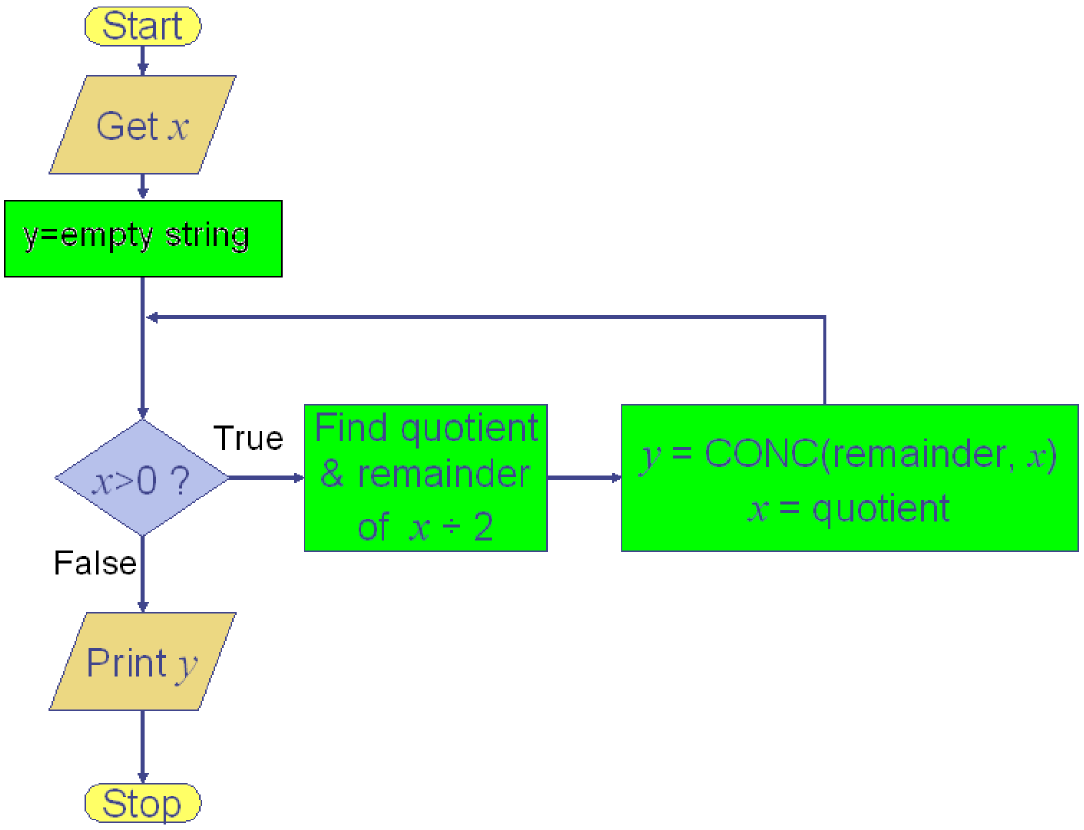
Step 1: Let x is the decimal integer you want to convert and let k=1

Step 2 : Divide x by 2, saving the quotient as Q, and the remainder (in binary) as R_k

Step 3 : If Q is not zero, let X=Q, and go back to step 2. Otherwise go to step 4.

Step 4 : Assume step 1-3 were repeated n times. Arrange the remainders as string for digit $R_n R_{n-1} ... R_3 R_2 R_1$.

b. As a flowchart



Flowchart of the algorithm of converting an integer from decimal to binary

c. By pseudocode

```
BEGIN
input x.
y=””
remainder=0,
  while (x>0)
begin
  quotient=x/2
  remainder=x mod 2
  y=conc(remainder,y)
  x=quotient
```

```
    end
    print y
    END.
```

Example

Bubble Sort

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

5 1 4 2 8 - unsorted array

1 4 2 5 8 - after one pass

1 2 4 5 8 - sorted array

The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps.Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

Here are the presentations of bubble sort algorithm

a. By words

Step 1: Get the length of the list : N and the list: list[1],list[2],…,list[N]

Step 2: M ← N.

Step 3: If M < 2 then print the list, stop.

Step 4: M ← M – 1, i ← 0.

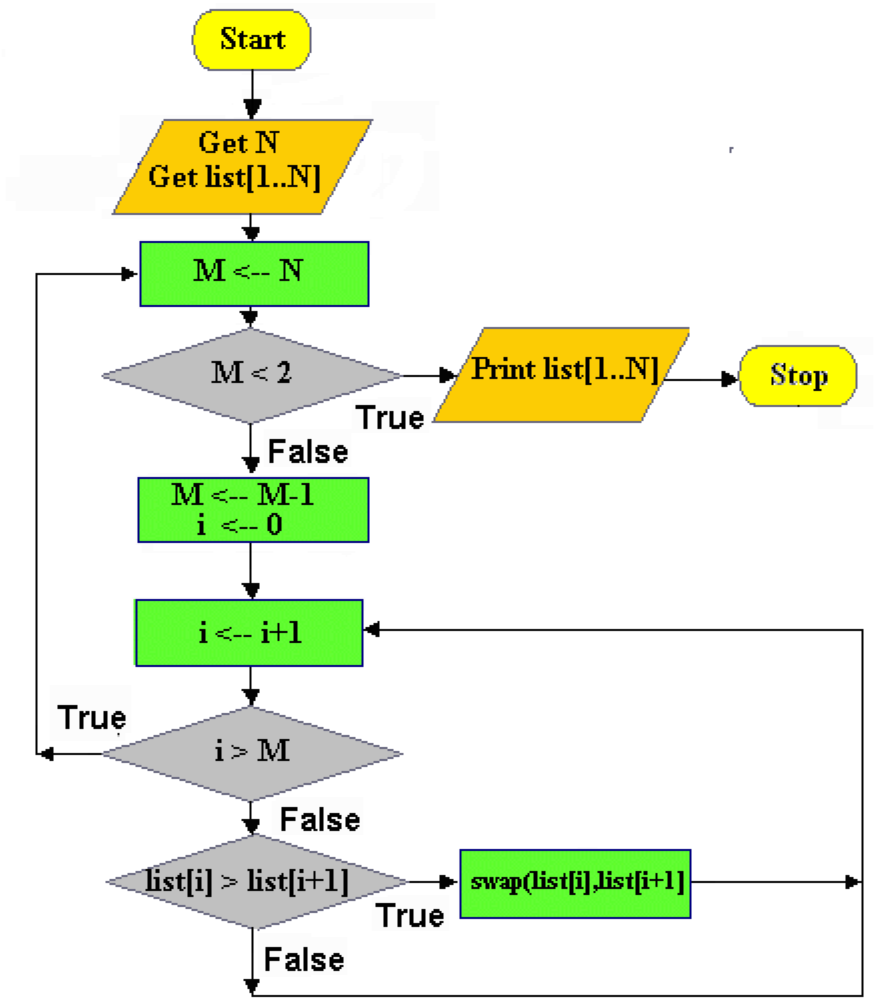
Step 5: Increase i by 1

Step 6: If i > M then go to step 3.

Step 7: If list[i] > list[i+1] swap list[i] and list[i+1]

Step 8: Go to step 5.

b. As a flow chart



Flowchart of bubble sort algorithm

c. In pseudocode

A simple way to express bubble sort in pseudocode is as follows:

```
BEGIN  get length (list) and list’s elements
  for each M in length(list) down to 2 do:
     for each i  in 1 to M-1 do:
       if list[ i] > list[ i+1 ] then
         swap( list[i+1],  list[ i ] )
       end if
     end for
  end for
end procedure
```

Comparing the three methods,especially pseudocode and flowchart we realized :

Pros and Cons of Flowcharts

In fact, flowcharts are not very useful.The process of writing an algorithm in the form of a flowchart is just too cumbersome, and then converting this graphical form into code is not straight forward

However, there is another kind of flowcharts – called Structured Flowcharts – that may be better suited for software developers.

The good thing about flowcharts is that their symbols are quite intuitive and almost universally understood. Their graphical nature makes the process of explaining an algorithm to one’s peers quite straightforward.

Pros and Cons of Pseudocode

Pseudocode are quite suitable for software development as it is closer in form to real code.One can write the pseudocode, then use it as a starting point or outline for writing real code.

Many developers write the pseudocode first and then incrementally comment each line out while converting that line into real code.Pseudocode can be constructed quite quickly as compared with a flowchart.

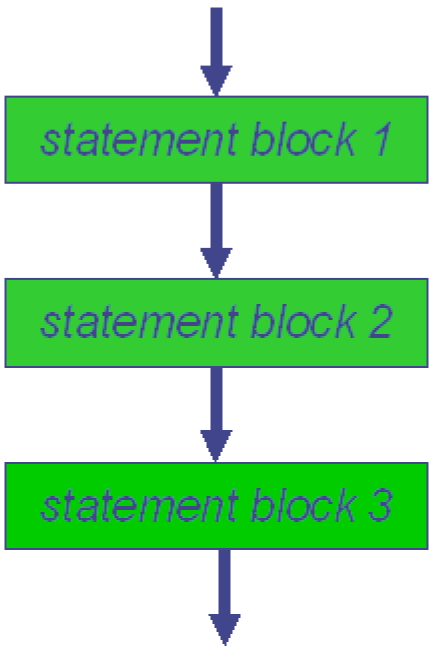
Unlike flowcharts, no standard rules exist for writing pseudocode

To design an algorithm, the following characteristics are very

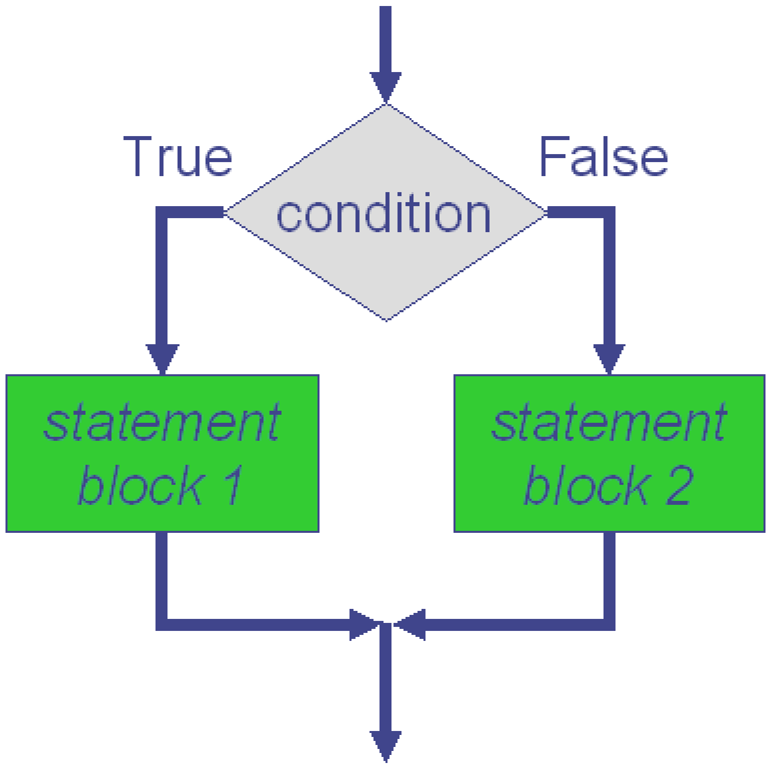
- Exactness
- Effectiveness
- Guaranteed termination
- Generality

The concept of structured programming says that any programming logic problem can be solved using an appropriate combination of only three programming structures,

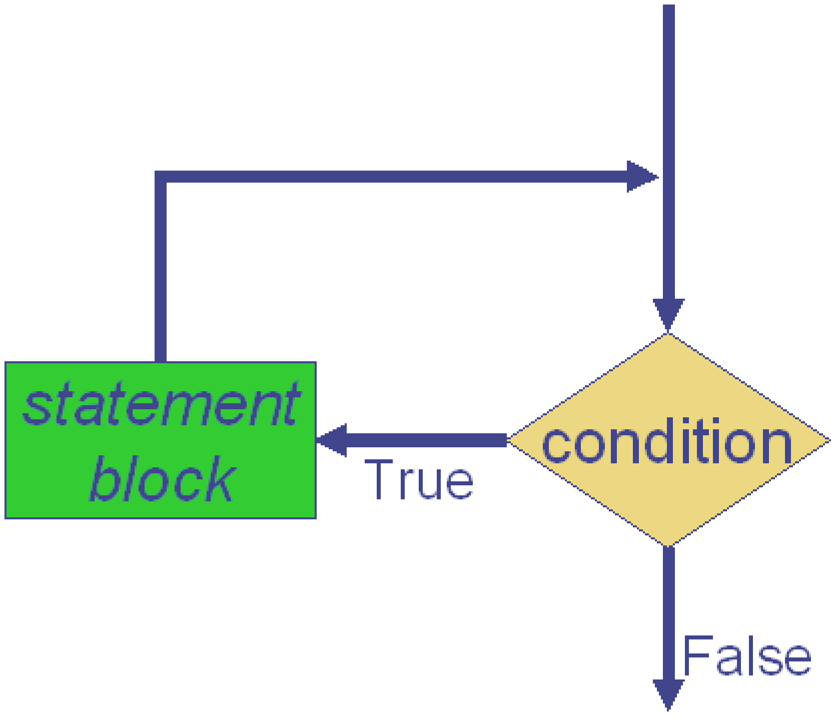
1.Sequence: a sequence of instructions that are executed in the precise order they are written in



2. Conditional : Select between alternate courses of action depending upon the evaluation of a condition



Loops: Loop through a set of statements as long as a condition is true



Programs and Programming Languages

Programs

A computer program is an algorithm written for a computer in a special programming language.

Programming languages

A programming language is an artificial language that can be used to control the behavior of a machine, particularly a computer. It is defined through the use of syntactic and semantic rules, to determine structure and meaning respectively.

Programming languages are used to facilitate communication about the task of organizing and manipulating information, and to express algorithms precisely.

There are large number of programming language in use. We can identify three type of programming languages : machine languages, assembly languages, high-level languages.

Machine Languages

Machine code or machine language is a system of instructions and data directly executed by a computer's central processing unit. Machine code is the lowest-level of abstraction for representing a computer program.Instructions are patterns of bits with different patterns corresponding to different commands to the machine. Machine code has several significant disadvantages : very difficult for a human to read and write, a program written on one computer cannot run on a different computer, so it cannot be used to write large program or program intended to run on different machines.

Assembly Languages

An assembly language is a low-level language for programming computers. It implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture.

This representation is usually defined by the hardware manufacturer, and is based on abbreviations (called mnemonics) that help the programmer remember individual instructions, registers, etc. An assembly language is thus specific to a certain physical or virtual computer architecture

A utility program called an assembler, is used to translate assembly language statements into the target computer's machine code.

Although assembly is more friendly than machine code, use of assembly offer several disadvantages, for instance, each type of computer has its own assembly language or programming assembly requires much time and effort.

Hence, assembly language is not use to write large programs. However, there are some computer application, such as in writing program that control peripherals, assembly is still a necessity.

High-level languages

A high-level programming language is a programming language that, may be more abstract, easier to use, or more portable across platforms.

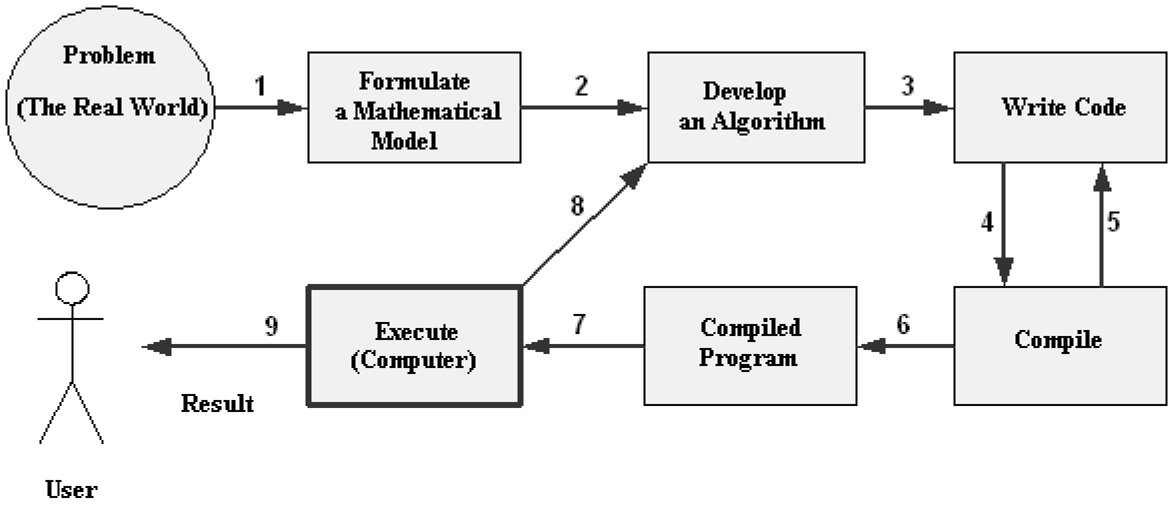
Examples: Pascal, C, Visual Basic, SQL, ...

Such languages often abstract away CPU operations such as memory access models and management of scope.These languages have been implemented by translating to machine languages.

There are two types of translators

- **Compiler** is a program that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine language)
- **Interpreter** is a program that translates and executes source language statements one line at a time.

[link] below shows the process of solving problem with computers



Steps in software development

Domain Analysis

Often the first step in attempting to design a new piece of software, whether it be an addition to an existing software, a new application, a new subsystem or a whole new system, is, what is generally referred to as "Domain Analysis". The more knowledgeable they are about the domain already, the less the work required. Another objective of this work is to make the analysts who will later try to elicit and gather the requirements from the area experts or professionals, speak with them in the domain's own terminology and to better understand what is being said by these people. Otherwise they will not be taken seriously. So, this phase is an important prelude to extracting and gathering the requirements.

Software Elements Analysis

The most important task in creating a software product is extracting the requirements. Customers typically know what they want, but not what software should do, while incomplete, ambiguous or contradictory requirements are recognized by skilled and experienced software engineers. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Specification

Specification is the task of precisely describing the software to be written, possibly in a rigorous way. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable.

Software architecture

The architecture of a software system refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.

Implementation (or coding)

Reducing a design to code may be the most obvious part of the software engineering job, but it is not necessarily the largest portion.

Testing

Testing of parts of software, especially where code by two different engineers must work together, falls to the software engineer.

Documentation

An important (and often overlooked) task is documenting the internal design of software for the purpose of future maintenance and enhancement. Documentation is most important for external interfaces.

Classification of Computer Software

The software is divided to System Software and Application Software with each having several sub levels.

System software is the low –level software required to manage computer resources and support the production or execution of application program.

Application software is software program that perform a specific function directly for the end user.

System Software includes

- Operating Systems software
- Network Software : network management software, server software, security and encryption software, etc.
- Database management software

- Development tools and programming language software: software testing tools and testing software, program development tools, programming languages software
- Etc.

Application Software includes

- General business productivity applications : software program that perform a specific function directly for the end user, examples include : office applications, word processors, spreadsheet, project management system ,etc.
- Home use applications : software used in the home for entertainment, reference or educational purposes, examples include games, home education etc.
- Cross-industry application software : software that is designed to perform and/or manage a specific business function or process that is not unique to a particular industry, examples include professional accounting software, human resources management, Geographic Information Systems (GIS) software, etc.
- Vertical market application software : software that perform a wide range of business functions for a specific industry such as manufacturing, retail, healthcare , engineering, restaurant, etc.
- Utilities software : a small program that performs a very specific task. Examples include : compression programs, antivirus, search engines, font, file viewers, voice recognition software, etc.

Operating Systems

Basic concepts

An operating system (OS) is the software that manages the sharing of the resources of a computer and provides programmers with an interface used to access those resources. An operating system processes system data and user input, and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system. At the foundation of all system software, an operating system performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems. Most operating systems come with an application that provides a user interface for managing the operating system, such as a command line interpreter or graphical user interface. The operating system forms a platform for other system software and for application software.

The most commonly-used contemporary desktop operating system is Microsoft Windows, with Mac OS X also being well-known. Linux and the BSD are popular Unix-like systems.

The operating system is the first thing loaded onto the computer -- without the operating system, a computer is useless. In detail, important services that an operating system provides are:

- Create Interface between you and your computer
- Manage the file system includes directories, folders, files

- Has a set of commands that allow for manipulation of the file system: sort, delete, copy, etc.
- Perform input and output on a variety of devices
- Allocate Resources
- Manage the running systems

Categorization of operating systems

All desktop and laptop computers have operating systems. Operating systems are categorized based on the types of computers they control and the sort of applications they support.

- Single-user, single task
- Single-user, multi-tasking
- Multi-user
- Real-time operating system

File

A computer file is a block of arbitrary information, or resource for storing information, which is available to a computer program and is usually based on some kind of durable storage. A file is durable in the sense that it remains available for programs to use after the current program has finished. Computer files can be considered as the modern counterpart of paper documents which traditionally were kept in offices' and libraries' files, which are the source of the term.

A filename is a special kind of string used to uniquely identify a file stored on the file system of a computer. Depending on the operating system, such a name may also identify a directory. Different operating systems impose different restrictions regarding length and allowed characters on filenames.

Many operating systems, including MS-DOS, Microsoft Windows, allow a filename extension that consists of one or more characters following the last period in the filename, thus dividing the filename into two parts: the base name (the primary filename) and the extension (usually indicating the file type associated with a certain file format). The base name and the extension are separated by a dot.

Commonly, the extension indicates the content type of the file, for example,

exe: executable file, txt : text file, pas : pascal source file, cpp : C++ source file. . .

In MS-DOS, Microsoft Windows, you can use wildcards ? and *. ? marks a single character while * Marks any sequence of characters.

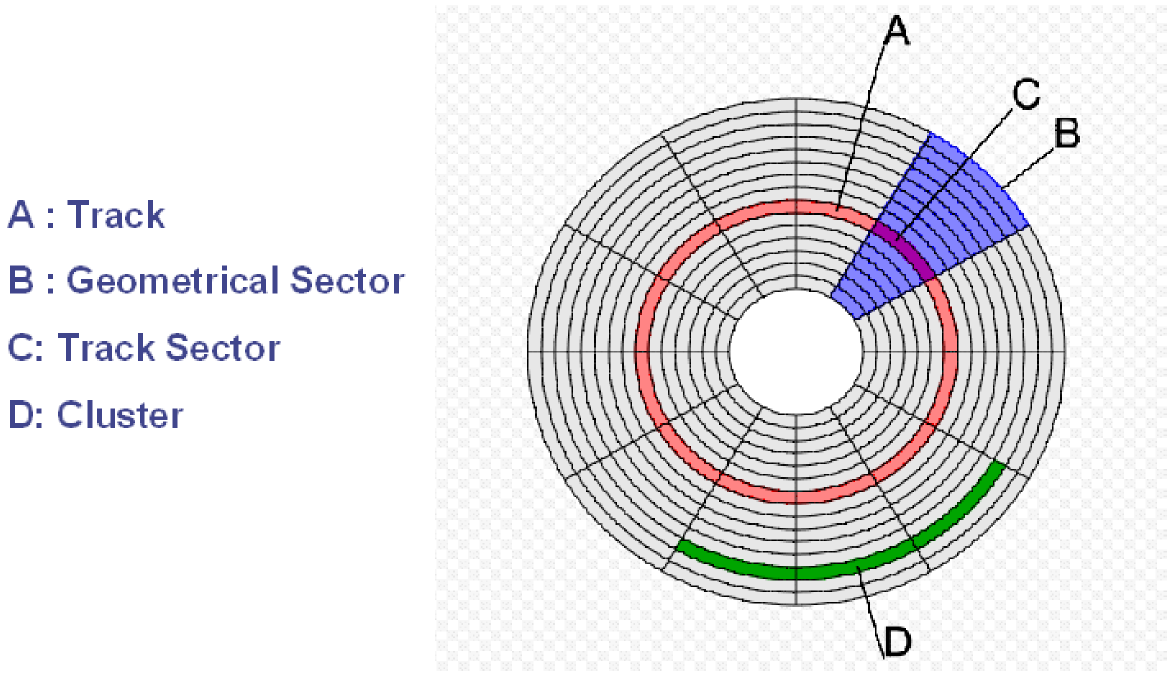
Example *.pas : all pascal source files of the current directory , possibly t1.pas, book.pas. . .

b*.cpp : all C++ source files beginning with b.

File Management

Structures of Disks

Floppy disk can be single-sided or double-sided. Data is stored on a disk in circular tracks. Tracks are numbered 0, 1. . . Each track is broken up into arcs called sectors. Each sector stores a fixed amount of data. The typical formatting of these media provide space for 512 bytes (for magnetic disks) or 2048 bytes (for optical discs) of user-accessible data per sector.



Track, sector, cluster

Formatting (initializing) a disk

Disk formatting is the process of preparing a hard disk or other storage medium for use, including setting up an empty file system. Formatting a disk includes the following tasks:

- Determines the sector size and placement.
- Slices the disk into sectors by writing codes on the disk.
- Locates bad spots on the disk, locks it out to prevent the bad spot from being used.
- Side number, track number, sector number Þ address : locates where on the disk the computer will store the data.

Computer file system

In computing, a file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, or they may be virtual and exist only as an access method for virtual data.

More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.

A typical file system may contain thousands (or even hundreds of thousands) of directories. Directory (catalog, or folder) is an entity in a file system which contains a group of files and/or other directories.Files are kept organized by storing related files in the same directory. A directory contained inside another directory is called a subdirectory of that directory. Together, the directories form a hierarchy, or tree structure.

https://cnx.org/content/m30793/

Directory tree

The first or top-most directory in a hierarchy is the root directory (symbolized by the back slash \)

The current directory is the directory in which a user is working at a given time.

Full name of a file

A full filename includes one or more of these components

- Drive (e.g., C:)
- Directory (or path) file
- Base name of the file
- Extension

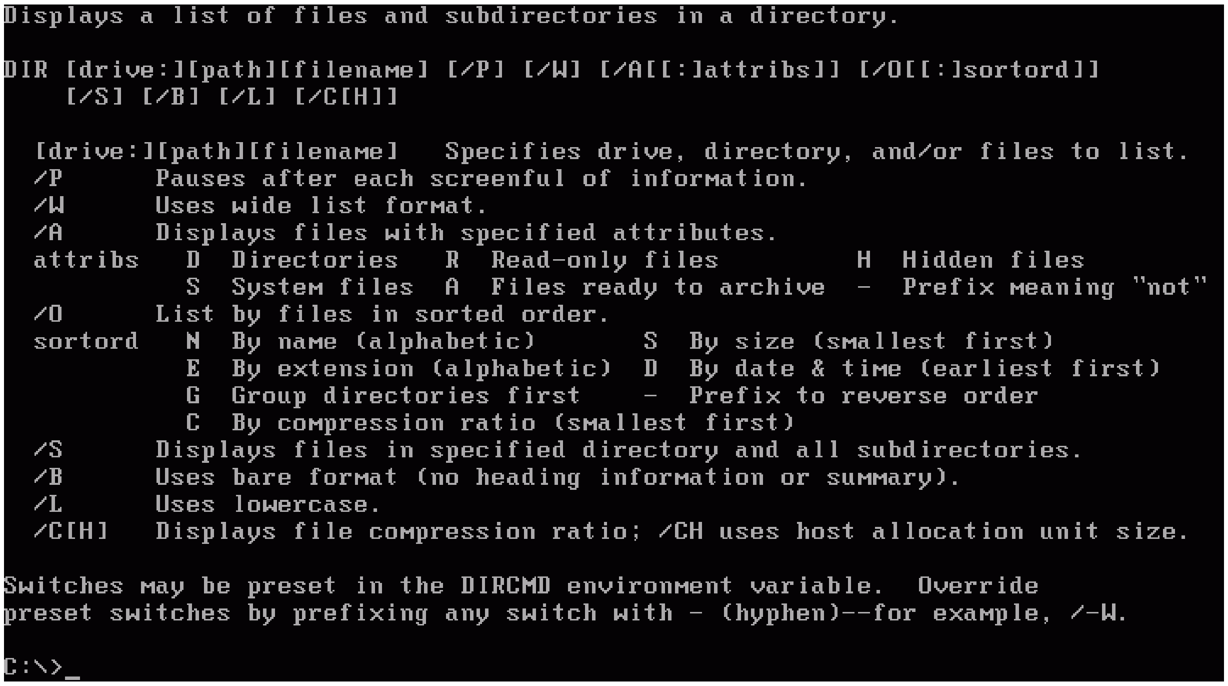
An operating system includes several files, for instant, MS-DOS includes MSDOS.SYS, IO.SYS, COMMAND.COM . . .

Some Common Operating Systems

MS-DOS

MS-DOS (short for Microsoft Disk Operating System) is an operating system commercialized by Microsoft. It was the most commonly used member of the DOS family of operating systems and was the dominant operating system for the PC compatible platform during the 1980s. It has gradually been replaced on consumer desktop computers by various generations of the Windows operating system.

MS-DOS employs a command line interface and a batch scripting facility via its command interpreter, COMMAND.COM.



```
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/P] [/W] [/A[[:]attribs]] [/O[[:]sortord]]
    [/S] [/B] [/L] [/C[H]]

  [drive:][path][filename]   Specifies drive, directory, and/or files to list.
  /P      Pauses after each screenful of information.
  /W      Uses wide list format.
  /A      Displays files with specified attributes.
  attribs   D  Directories   R  Read-only files         H  Hidden files
            S  System files  A  Files ready to archive  -  Prefix meaning "not"
  /O      List by files in sorted order.
  sortord   N  By name (alphabetic)       S  By size (smallest first)
            E  By extension (alphabetic)  D  By date & time (earliest first)
            G  Group directories first    -  Prefix to reverse order
            C  By compression ratio (smallest first)
  /S      Displays files in specified directory and all subdirectories.
  /B      Uses bare format (no heading information or summary).
  /L      Uses lowercase.
  /C[H]   Displays file compression ratio; /CH uses host allocation unit size.

Switches may be preset in the DIRCMD environment variable.  Override
preset switches by prefixing any switch with - (hyphen)--for example, /-W.

C:\>_
```

The MS-DOS 6.22 command line interface

Microsoft Windows

Microsoft Windows is the name of several families of software operating systems by Microsoft. Microsoft Windows interest in graphical user interfaces (GUI)

MsWindows are introduced in detail in the next section.

The Most Common Commands of an Operating Systems

Every operating system need a system of command for managing files and disks. Commonly used types are :

- File management : copy, delete, rename, type a file.

- Directories management: create, remove, copy directories.
- Disk management : disk copy, disk format.

Microsoft Windows

Brief History of Microsoft Windows

In 1983 Microsoft announced its development of Windows, a graphical user interface (GUI) for its own operating system. Windows 3.0, released in 1990, was a complete overhaul of the Windows environment with the capability to address memory beyond 640K and a much more powerful user interface.

Windows for Workgroups 3.1 was the first integrated Windows and networking package offered by Microsoft.Windows for Workgroups also includes two additional applications: Microsoft Mail, a network mail package, and Schedule+, a workgroup scheduler.

Windows 95, released in August of 1995. A 32-bit system providing full pre-emptive multitasking, advanced file systems, threading, networking and more.Also includes a completely revised user interface.

Windows 98, released in June of 1998. Integrated Web Browsing gives your desktop a browser-like interface.

Windows 2000 provides an impressive platform of Internet, intranet, extranet, and management applications that integrate tightly with Active Directory.

In September 2000 Microsoft released Windows Me, short for Millennium Edition, which is aimed at the home user. The Me operating system boasts some enhanced multimedia features, such as an automated video editor and improved Internet plumbing.

Windows XP-Microsoft officially launches it on October 25th. 2001.XP is a whole new kind of Windows for consumers. Under the hood, it contains the 32-bit kernel and driver set from Windows NT and Windows 2000.

Naturally it has tons of new features that no previous version of Windows has.

Windows Vista is a line of graphical operating systems used on personal computers, including home and business desktops, notebook computers, Tablet PCs, and media centersWindows Vista contains hundreds of new and reworked features; some of the most significant include an updated graphical user interface and visual style dubbed Windows Aero, improved searching features, new multimedia creation tools such as Windows DVD Maker, and completely redesigned networking, audio, print, and display sub-systems.

Originally developed as a part of its effort to introduce Windows NT to the workstation market, Microsoft released Windows NT 4.0, which features the new Windows 95 interface on top of the Windows NT kernel.

Windows NT (New Technology) is a family of operating systems produced by Microsoft, the first version of which was released in July 1993. It was originally designed to be a powerful high-level-language-based, processor-independent, multiprocessing, multiuser operating system with features comparable to Unix. It was intended to complement consumer versions of Windows that were based on MS-DOS. NT was the first fully 32-bit version of Windows, whereas its consumer-oriented counterparts, Windows 3.1x and Windows 9x, were 16-bit/32-bit hybrids. Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008 (beta), and Windows Home Server are based upon the Windows NT system, although they are not branded as Windows NT.

Windows XP is the most popular version of Microsoft Windows. Windows provides a graphical interface, through which you can run programs, manage files, connect to the internet, and perform many other task as well.

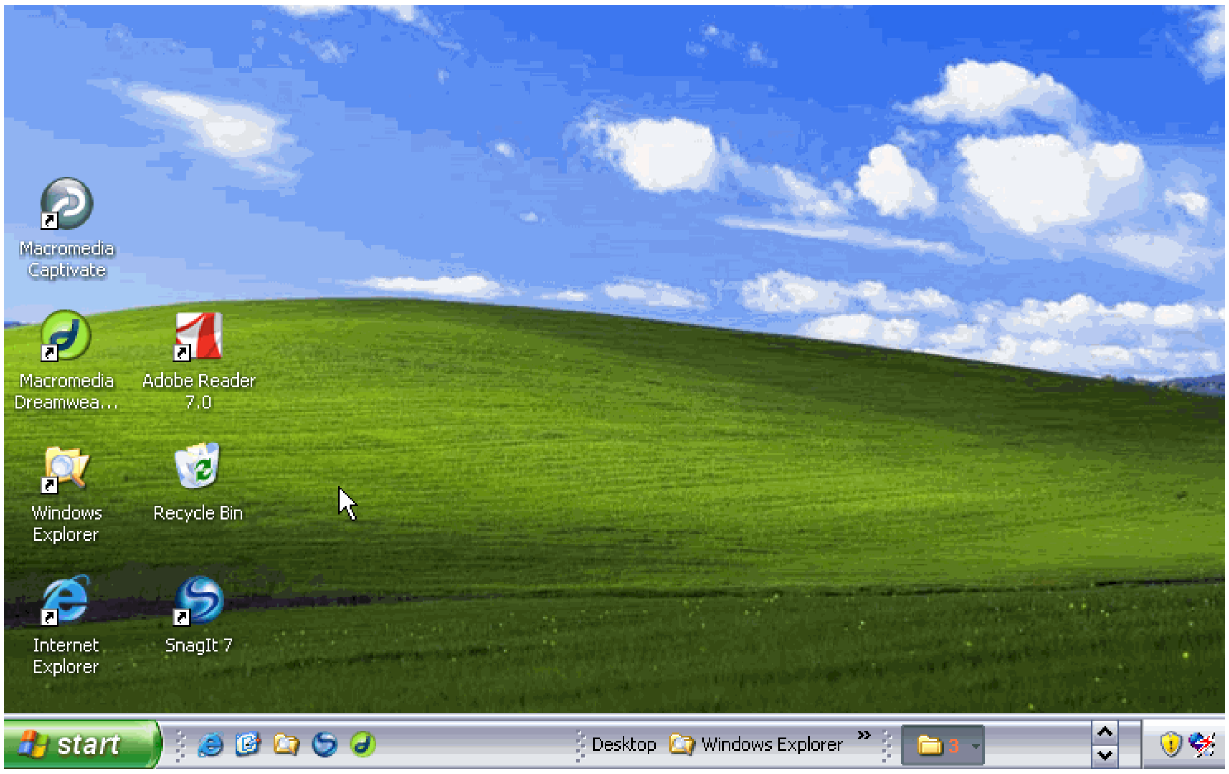
How to start and exit from Windows XP

Starting Windows XP

Windows XP starts automatically when you turn on your computer. Depending on the way your PC is currently set up, you may be prompted to select a user account when you start up your PC. Windows will display a welcome screen, from which you click your user name and indicate who you are by entering your password.

Once Windows XP has initialized, the following screen will appear.

Each user has his own ideas about what constitutes attractive screen colors, important shortcut to place on the desktop etc. This combination can be saved as user profile and Windows remembers all the setting and preferences.



Shutting down Windows XP

When you finished using your PC, you shouldn't turn off the power because that could cause later problems in Windows. Instead, you should use the

Shut Down command on the Start menu (or press Ctrl+Esc if the Start menu is invisible). This approach ensure that Windows shuts down in an orderly way that closes all opened files and saves your work in any open program.

When shutting down, you have two options: Turn Off and Restart. If you are probably to be away from the computer, you will probably want to turn it off. If the computer is acting strangely and you want to start fresh, you will want to refresh.

If for some reason, the computer is not ready to shut down , the computer will remind you in dialog boxes.

Basic Terms and Operations

The Icons

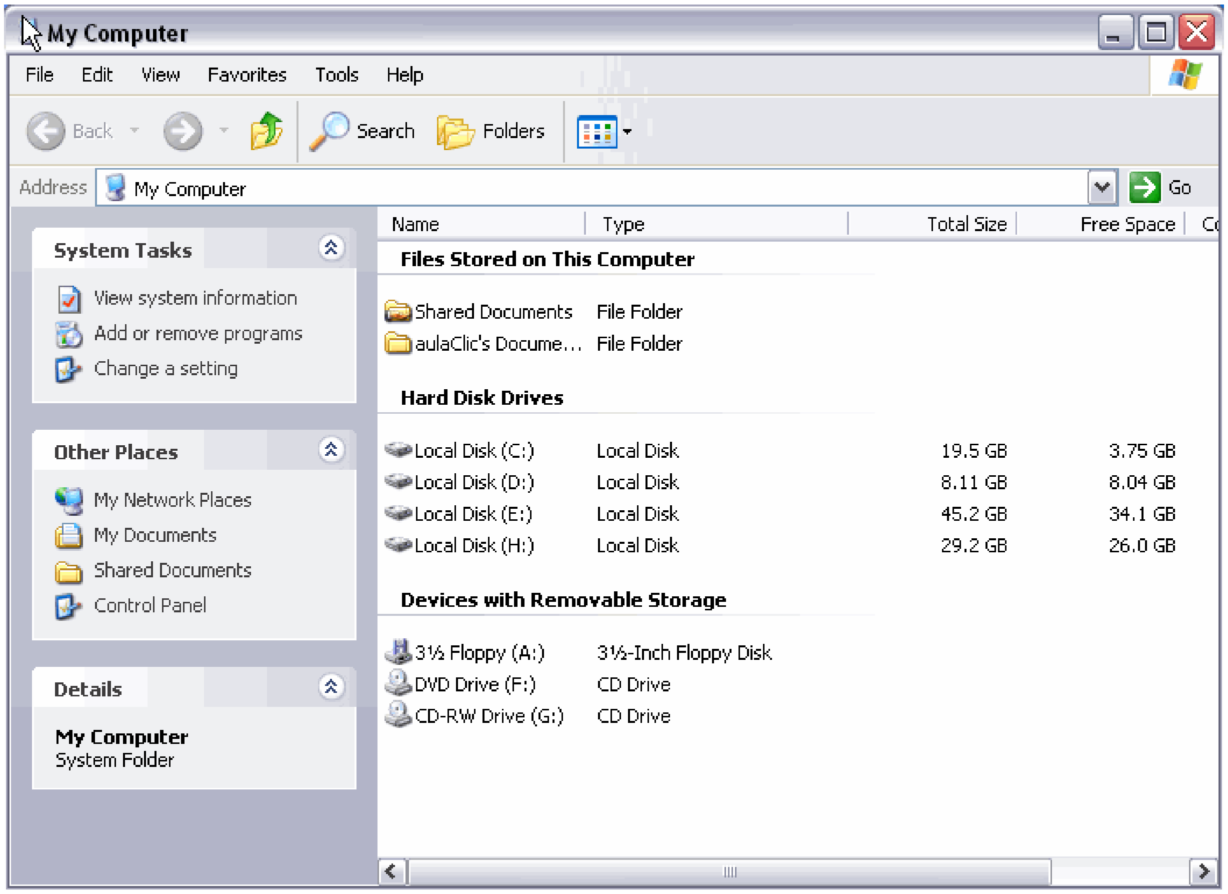
On the desktop we have icons that allow us to open the corresponding program.

For example, by clicking on the icon

Internet Explorer will open up.

The windows

All the windows have the same structure;The window above is the one that opens when you click on My Computer. Its structure is very similar to the others.



My Computer
System Folder

Name	Type	Total Size	Free Space
Files Stored on This Computer			
Shared Documents	File Folder		
aulaClic's Docume…	File Folder		
Hard Disk Drives			
Local Disk (C:)	Local Disk	19.5 GB	3.75 GB
Local Disk (D:)	Local Disk	8.11 GB	8.04 GB
Local Disk (E:)	Local Disk	45.2 GB	34.1 GB
Local Disk (H:)	Local Disk	29.2 GB	26.0 GB
Devices with Removable Storage			
3½ Floppy (A:)	3½-Inch Floppy Disk		
DVD Drive (F:)	CD Drive		
CD-RW Drive (G:)	CD Drive		

All the windows are formed by:

- The title bar contains the name of the program you are working with and in some cases the name of the opened document also appears. In the top right corner we can find the minimize, maximize/restore, and close buttons.
- The minimize

  button shrinks the window it turns it into a button located in the WindowsXP task bar.
- The maximize

  amplifies the size of the window to the whole screen.

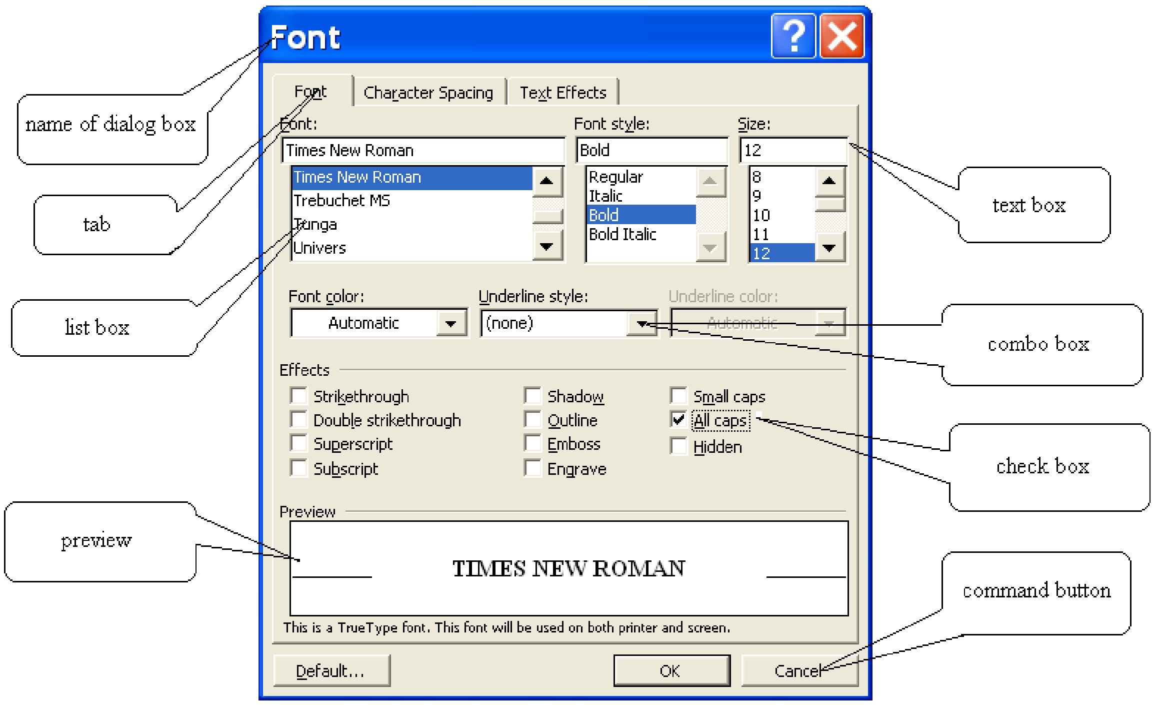
- The restore button

  restores the window to its original state.
- The close button

  closes the window. If we have modified the document, we are asked if we want to save the changes before closing.

The dialog boxes

The dialog box is a small window-like box that opens after an operation has been selected. In it, you select options and settings to tailor the operation before it proceeds.



Font: Times New Roman (Times New Roman, Trebuchet MS, Tunga, Univers)

Font style: Bold (Regular, Italic, Bold, Bold Italic)

Size: 12 (8, 9, 10, 11, 12)

Font color: Automatic

Underline style: (none)

Underline color: Automatic

Effects:

- [ ] Strikethrough
- [ ] Double strikethrough
- [ ] Superscript
- [ ] Subscript
- [ ] Shadow
- [ ] Outline
- [ ] Emboss
- [ ] Engrave
- [ ] Small caps
- [✓] All caps
- [ ] Hidden

TIMES NEW ROMAN

This is a TrueType font. This font will be used on both printer and screen.

Text box : a control in which a user can enter texts (or numbers).

List box : a box that contains a list of selectable items. In some instances, you select an arrow button on the right of the box in order to display the selectable items.

Combo box : a combination of a drop-down list or list box and a single-line textbox, allowing the user either to type a value directly into the control or choose from the list of existing options.

Check box : a control that permits the user to make single selection or multiple selections from a number of options. Normally, check boxes are shown on the screen as a square box that can contain white space (for false) or a tick mark or X (for true).

Command Button: A control used to initiate an action. The most common buttons are :

- OK
- Close
- Cancel
- Apply
- Default

Using a computer mouse

Use the mouse to interact with items on your screen as you would use your hands to interact with objects in the physical world. You can move objects, open them, change them, or throw them away, among other things.

A mouse has a primary and secondary mouse button. Use the primary mouse button to select and click items, position the cursor in a document, and drag items.

Use the secondary mouse button to display a menu of tasks or options that change depending on where you click. This menu is useful for completing tasks quickly. Clicking the secondary mouse button is called right-clicking.

The primary mouse button is normally the left button on the mouse. On a trackball, the primary mouse button is normally the lower button.

You can reverse the buttons and use the right mouse button as the primary button.Most mice now include a wheel that helps you to scroll through documents more easily.

Pointing

Pointing at items on the screen is the most basic mouse function. When instructions tell you to point your mouse at something, move your mouse on your desk until the mouse pointer is pointing at the object on the screen you need to select.

Clicking

After you have pointed your mouse at an item, you can click on the item to select it.

Double clicking

To double-click an item, point at the item and press your primary button twice quickly without moving the mouse. Double-clicking allows two different actions to be associated with the same mouse button. Often, single-clicking selects (or highlights) an object, while a double-click executes that object, but this is not universal.

Drag and drop

to move the item from one place to another using the mouse. Point at the item you need to move, and single click on it. Instead of releasing the mouse button after clicking, hold it down, and move your mouse to where you want to move the item. Release the mouse button to drop the item into place.

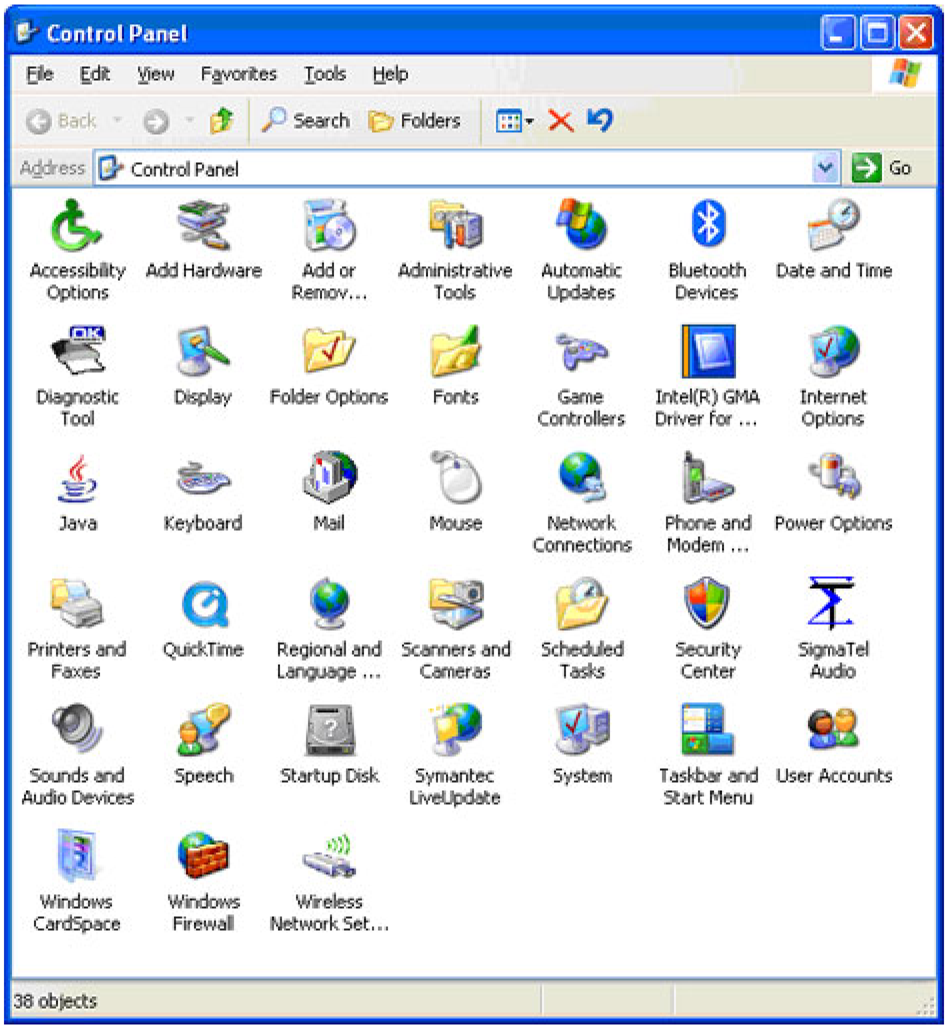
Right clicking

Right-clicking an item usually brings up a menu of actions you can take with the item. To right-click, point at an item and press the secondary (right) button on your mouse.

The Control Panel

Control Panel allows users to view and manipulate basic system settings and controls, such as adding hardware, adding and removing software, controlling user accounts, and changing accessibility options.

To start the Control Panel, from the Start menu, click on Control Panel. Here is the Control Panel window:

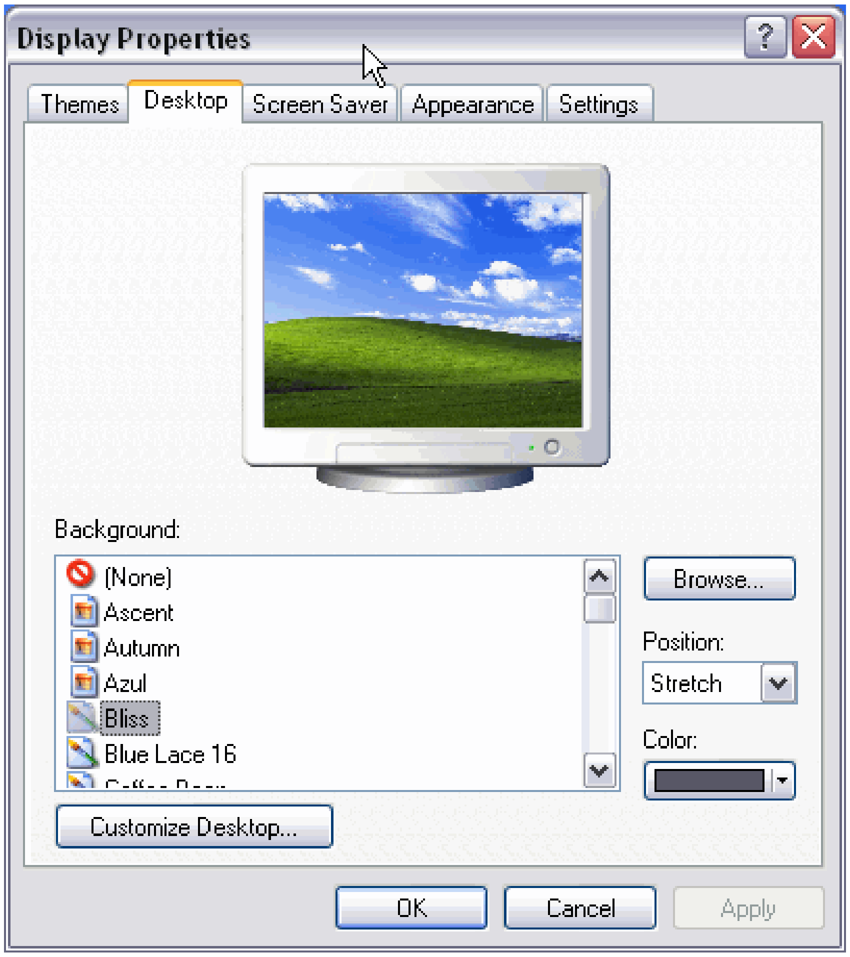


.

Configuring the Screen

Configuring the screen is important because sometimes we spend many hours in front of the screen, so we hope it can be the most comfortable as possible.

Open the Display Tool (or right-click somewhere that has no icons on the desktop and select the option Properties from the shortcut menu that is displayed. The Display properties window will appear where we can change the configuration parameters.



Change the background or wallpaper,

Click on the tab labeled Desktop and choose a new background or wallpaper from the list that appears at the bottom left corner. It is also possible to have another image that does not appear on the list as background. Click on Browse... and look for the image you want as long as the format is compatible. For example .bmp, .jpg, .gif.Once the image and type of view have been selected Click OK.

The screensaver

Sometimes the computer remains inactive a few minutes. It is recommended to have a screensaver to avoid having a still image on the screen too long because this can damage the monitor.

From the list, choose the screensaver you like best; a small preview is shown above.

You can modify the time it takes for the screensaver to appear by adjusting the time on Wait.

Configuring the Mouse

The mouse is a tool that is used constantly and it is recommendable to have it set up to our needs as well as possible. In the following page we show you how to set it up to your own needs.

The Buttons

On the Buttons tab you can adjust the set up of the mouse to suit your needs.If you are left handed. WindowsXP allows you to change the configuration of the buttons so that the right button realizes these functions.

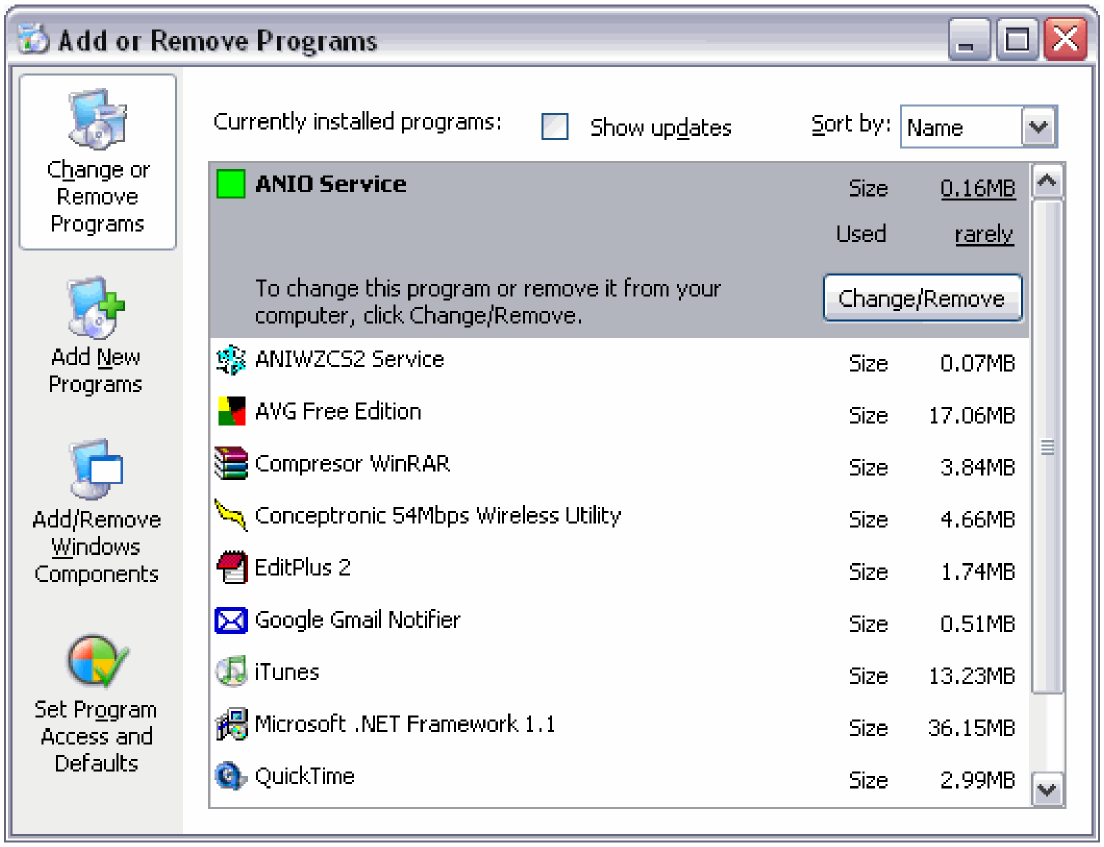
We can also adjust the Double-click speed for a slower or a faster double-click.

The pointer

On the Pointers tab we can choose the type of pointer the mouse is to have when it moves, when it is busy, when it is used, etc.

Adding or removing Programs

- Click on the Start button and choose Control Panel
- Click on Add or Remove Programs option, a window will display with the three basic options shown on the left side of the picture as it appears below. Then click on Add New Programs.The window will appear where we can change the configuration parameters.
- Follow the instruction
- The Add or Remove Programs window will appear where we can add, change or remove programs following the instructions..


Add or Remove Programs
Change or Remove Programs
Add New Programs
Add/Remove Windows Components
Set Program Access and Defaults
Show updates
Change/Remove

Currently installed programs: ☐ Show updates   Sort by: Name

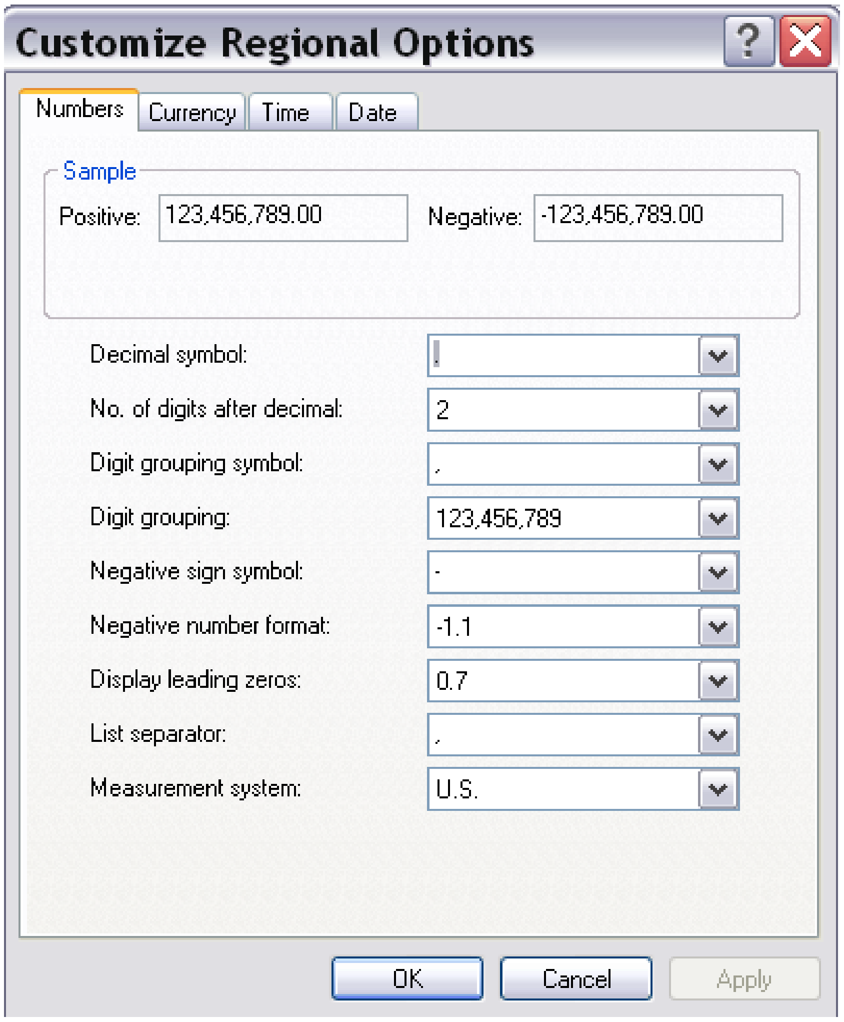
ANIO Service	Size	0.16MB
	Used	rarely
To change this program or remove it from your computer, click Change/Remove.		
ANIWZCS2 Service	Size	0.07MB
AVG Free Edition	Size	17.06MB
Compresor WinRAR	Size	3.84MB
Conceptronic 54Mbps Wireless Utility	Size	4.66MB
EditPlus 2	Size	1.74MB
Google Gmail Notifier	Size	0.51MB
iTunes	Size	13.23MB
Microsoft .NET Framework 1.1	Size	36.15MB
QuickTime	Size	2.99MB

Changing the Regional and Language Options

You can use the Regional and Language Options tool in Control Panel to customize the way Windows handles dates, times, currency values, and numbers.

To open the Regional and Language Options tool

- Click Start, and then click Control Panel.
- Click Date, Time, Language, and Regional Options, and then click Regional and Language Options.
- To change one or more of the individual settings, click Customize.



Positive: 123,456,789.00 Negative: -123,456,789.00

Decimal symbol:	.
No. of digits after decimal:	2
Digit grouping symbol:	,
Digit grouping:	123,456,789
Negative sign symbol:	-
Negative number format:	-1.1
Display leading zeros:	0.7
List separator:	,
Measurement system:	U.S.

To Change the Date In the Customize Regional Options dialog box, click the Date tab to specify any changes you want to make to the short date and the long date.

To Change the Time In the Customize Regional Options dialog box, click the Time tab to specify any changes you want to make.

To Change the Currency Value Display

In the Customize Regional Options dialog box, click the Currency tab to specify any changes you want to make. You can change the currency symbol, the formats used for positive or negative amounts, and the punctuation marks.

To Change the Number Display

In the Customize Regional Options dialog box, click the Numbers tab to specify any changes you want to make. You can change the decimal symbol and list separator, the format used for negative numbers and leading zeros, and the measurement system (U.S. or metric).

Add a printer

- Click on Printer and Faxes
- Click on Add a Printer, follow the instruction of the Add Printer Wizard

Delete a printer

- Click on Printer and Faxes
- Click on the printer you wish to delete.
- Press your Delete key to delete the printer.

The Windows Explorer

The Explorer is an indispensable tool in an operating system, since with it we can organize and control the files and folders of the different storage systems at our disposal such as the hard drive, disk drive, etc.

The Windows Explorer is also known as the File Manager. Through it we can delete, see, copy, or move files and folders.

Starting the Explorer

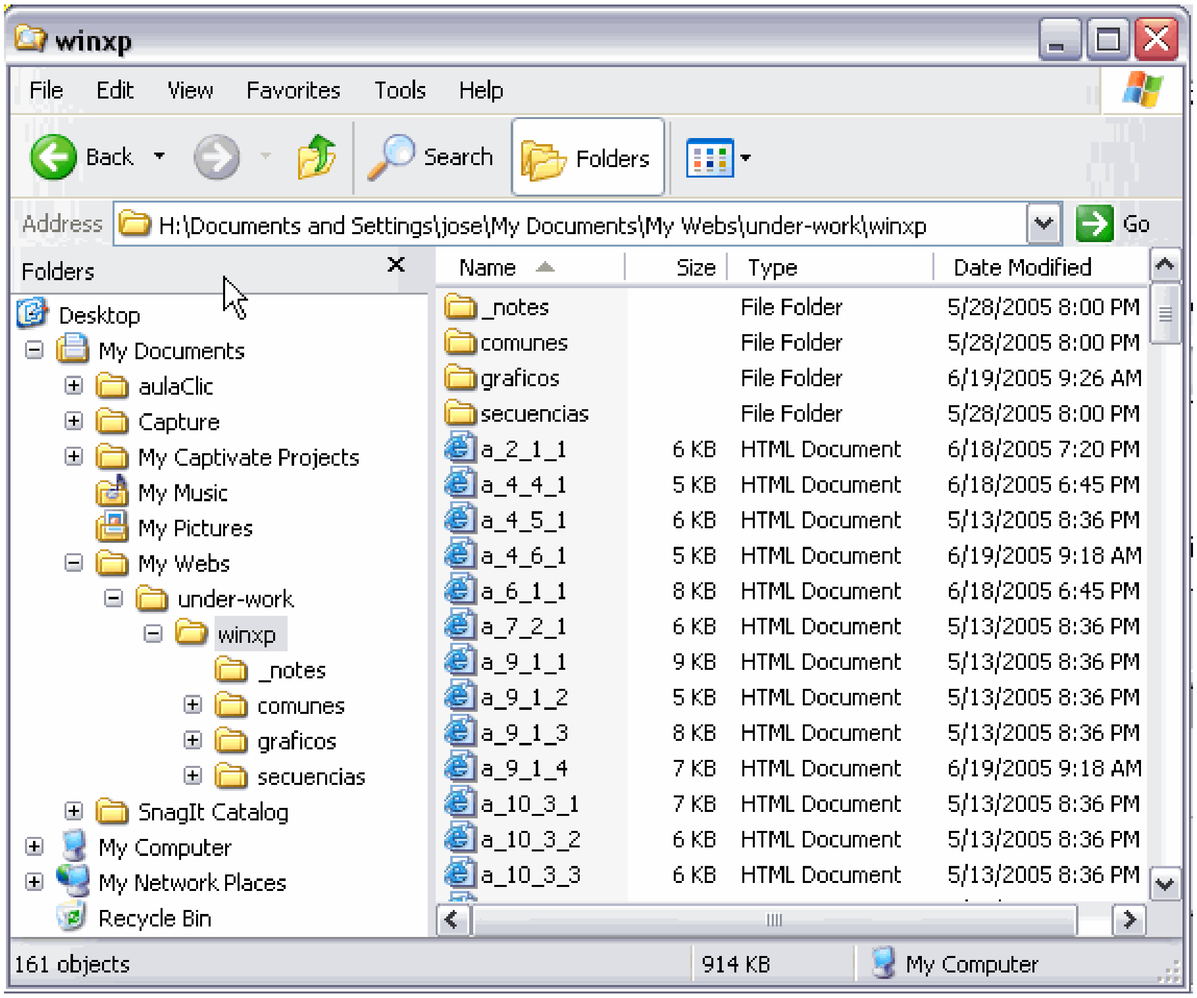
The quickest way to start up the Explorer is through the icon

on the task bar or desktop. If you don't already have the icon created, you can open the Explorer as follows:

- Click on Start   Select All programs   Select Accessories   Select Windows Explorer
- Right click on Start button and select Explore
- From the Start button, choose My documents, My images or My music; the difference is that in these cases we will go directly to those folders.

The Explorer's window

The explorer consists basically of two sections. On the left side there is the directory tree, which is the list of units and folders that we have. Only units and folders appear, no files. On this image we can see a few folders such as My Documents, aulaclic, ... the My Computer icon, My Network Places and the Recycle Bin.



H:\Documents and Settings\jose\My Documents\My Webs\under-work\winxp

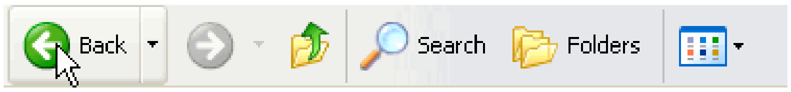
Desktop
- My Documents
  - aulaClic
  - Capture
  - My Captivate Projects
  - My Music
  - My Pictures
  - My Webs
    - under-work
      - winxp
        - _notes
        - comunes
        - graficos
        - secuencias
  - SnagIt Catalog
- My Computer
- My Network Places
- Recycle Bin

Name	Size	Type	Date Modified
_notes		File Folder	5/28/2005 8:00 PM
comunes		File Folder	5/28/2005 8:00 PM
graficos		File Folder	6/19/2005 9:26 AM
secuencias		File Folder	5/28/2005 8:00 PM
a_2_1_1	6 KB	HTML Document	6/18/2005 7:20 PM
a_4_4_1	5 KB	HTML Document	6/18/2005 6:45 PM
a_4_5_1	6 KB	HTML Document	5/13/2005 8:36 PM
a_4_6_1	5 KB	HTML Document	6/19/2005 9:18 AM
a_6_1_1	8 KB	HTML Document	6/18/2005 6:45 PM
a_7_2_1	6 KB	HTML Document	5/13/2005 8:36 PM
a_9_1_1	9 KB	HTML Document	5/13/2005 8:36 PM
a_9_1_2	5 KB	HTML Document	5/13/2005 8:36 PM
a_9_1_3	8 KB	HTML Document	5/13/2005 8:36 PM
a_9_1_4	7 KB	HTML Document	6/19/2005 9:18 AM
a_10_3_1	7 KB	HTML Document	5/13/2005 8:36 PM
a_10_3_2	6 KB	HTML Document	5/13/2005 8:36 PM
a_10_3_3	6 KB	HTML Document	5/13/2005 8:36 PM

161 objects | 914 KB | My Computer

On the right side there is another section, which will show the content of the folder that we have opened on the left section. This section shows its folders and files. In this case the files that are contained in the folder WinXP appear. Depending on the type of view that we have activated we will see different type of information regarding the files.

Next we will explain the different bars that make up this window.



The standard bar contains the buttons for the most used operations.

If this bar is not visible select from the menu View, the option Toolbars, next select the option Standard buttons.

The Back button

will allow us to go to the last page that we have seen. The button next to it, when activated, allows us to move one page forward.

The up button

will allow us to go up one level, which means going back to the folder that contains the folder we are working with.

The search button

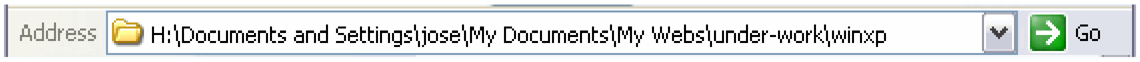
displays a window where we can search for the file we want.

The folders button

shows the folder's structure on the left side of the screen, or it can display an area with the most frequent tasks, depending on the file we have selected. In this area we can find, among others, the following buttons:

The last button

allows us to change the views on the folders (view details, Thumbnails,...) We'll explain this in more detailed on the next page.


Address
H:\Documents and Settings\jose\My Documents\My Webs\under-work\winxp
Go

The Address Bar is well known for Internet because it shows the address of the web we are viewing. With Windows Explorer it functions the same way, but it shows the name of the folder we are working with.

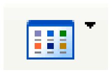
If we click on the black arrow it will show the structure with our computer's drives.

If we write a name in the address bar and we click on the green arrow, it will search for this name.

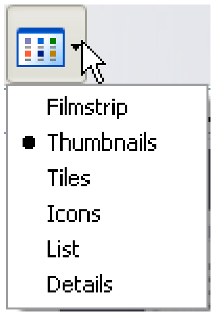
Windows explorer allows us to see the folder's information in different ways or views to facilitate specific searching.

Go to the folder you wish to see:

If you click on the arrow of the button



a menu with the following options will appear:



Tiles. The files and folders are shown with large images with the name, file type and size in KB; if it is a picture file the size is shown in pixels. The elements are organized one next to the other from left to right.

Icons. The files are represented with an icon smaller than a tile. The only information shown is the name of the file. This type of icon is used when the selected folder has an average quantity of elements.

List. Shows small icons, one below the other, so it's easier to search by name. On this view, only the name of the file or folder appears.

Details. Icons are shown one below the other, with some of their properties. This type of display is used when we want to find an element with certain characteristics, such as size, file type, date of modification, etc.

With this type of view we can organize the elements by size, modification date, name, etc.

For example, to organize by the modification date it is enough to click on the box Date Modified, and it will arrange the files by date from greater to lesser. If we click on it again it will arrange it from lesser to greater. The older dates are considered lesser.

On the views List or Details the elements appear one below the other and in the case of deleting or adding, the elements will reorganize themselves.

Thumbnails . A small representation of the content will appear with the format of the image, such as jpg., jpeg., bmp., gif., etc.

Filmstrip. This view is only available for images. On the bottom part a strip will appear with the images in thumbnail format and on the top we will see a larger representation of the image selected on the bottom.

Opening Files

Choose one of the following ways:

- Double click on the file’s icon.
- Right click on the file’s icon. Select Open
- Select the file and press Enter.

Selecting Files

If you wish to select a single file or folder you simply need to click on it. This way any operation you perform will only apply to the selected file or folder.

If you wish to realize an operation on several files or folders, Windows Explorer will allow you to select several elements at the same time.

To select consecutive elements

Click on the first element and then click on the last element while keeping Shift key pressed. This can also be done with the mouse. To do this, click on the left of the first element (but not on it) and, without letting go, drag it. A frame should appear that shows the area that the frame encompasses. Continue dragging until all the desired elements are within the frame, then let go of the left mouse button..

To select several elements that are not consecutive

Select the first element and continue to select the desired elements while keeping the Ctrl key pressed.

Creating and Deleting Folders

To create a folder we need to place the pointer where we want the folder to be.Open the folders that we have by clicking on the + located to the left of the folders.

If we click on the plus sign of a particular folder it will display and show all of the folders contained in it and the plus sign will become a minus sign -; this will take care of retracting the folders displayed, or hide the content of the folder selected.

Once we have the folder that we want open we will select it by clicking on the appropriate folder .Open the menu File, select the option New and then select the option Folder.

Now we can view on the bottom right window a new folder that has the name New Folder. This is the name that Windows gives new folders by default. In the event that it finds another folder with that same name, it will subsequently name the new folders New Folder(1), New Folder(2), etc...

The name of the folder can be changed

Deleting folders

To Delete a folder, first place the pointer on it.

Once the folder has been selected go to the Standard bar and click on or you can use Delete.

When we delete a folder or file, by default Windows will move it to the Recycle Bin. The settings can be changed so that it deletes it completely.

The Recycle Bin is nothing more than a space reserved on the hard disk so that in case of having deleted any element it would be possible for us to retrieve it.

Deleting Files

To delete a file we follow the same steps to delete a folder, but instead of selecting a folder select the file you wish to delete.

Copying Files or Folders

Select the element to be copied.Click on Copy and it will open a dialog box titled Copy Items. If we do not have this button on the tool bar, we can go to the Edit menu and select Copy to Folder... First select the item to copy

Search for the folder to which we will copy the selected element. It works like Windows explorer. If we click on the + that appears on the left, the contents of the folder will be displayed.

Once the folder has been selected, click on Copy.

In the case of not having the folder created to which we want to copy to, click Make new folder, write the name of the new folder and Click OK.

Moving Files or Folders

Moving a file or folder means copying the element to the desired location and then deleting its original location. The following steps are very similar.

- Select the file or folder you want to move.
- Click on , or Edit --> Move to Folder which will open a new window titled Move Items.
- Search for the folder where the element are to be moved to.
- Once the folder is selected, click Move.
- In the case of not having the folder created to which we want to move the information to, simply click Make New Folder.
- Write the name of the new folder.Click OK.

When moving or copying an item, its name can coincide with the name of a file or folder that is in the destination folder. In this case Windows will ask if we want to substitute the existing file or folder by the new one. When folder is moved or copied, its entire content is also moved or copied.

Changing the name of a File or Folder

- Select the file or folder that you want to change the name of.
- With the right mouse button click on it.

- Select Rename from the shortcut menu, then the name of the file or folder will be highlighted and with the pointer blinking inside the name box.
- Write the new name.
- Click Enter or click outside the file or folder so that the changes take place.
- You can also do this with Rename option from File menu.

Files and Folders Properties

Both files and folders have their own characteristics, for example size, location, date of creation, attributes, etc.

To know the characteristics of a particular file or folder we need to:

- select it and choose Properties option from File menu,or,
- click on it with the right mouse button and select the option Properties from the menu that is displayed.

Click on the OK button to accept or the Cancel button to discard all changes.

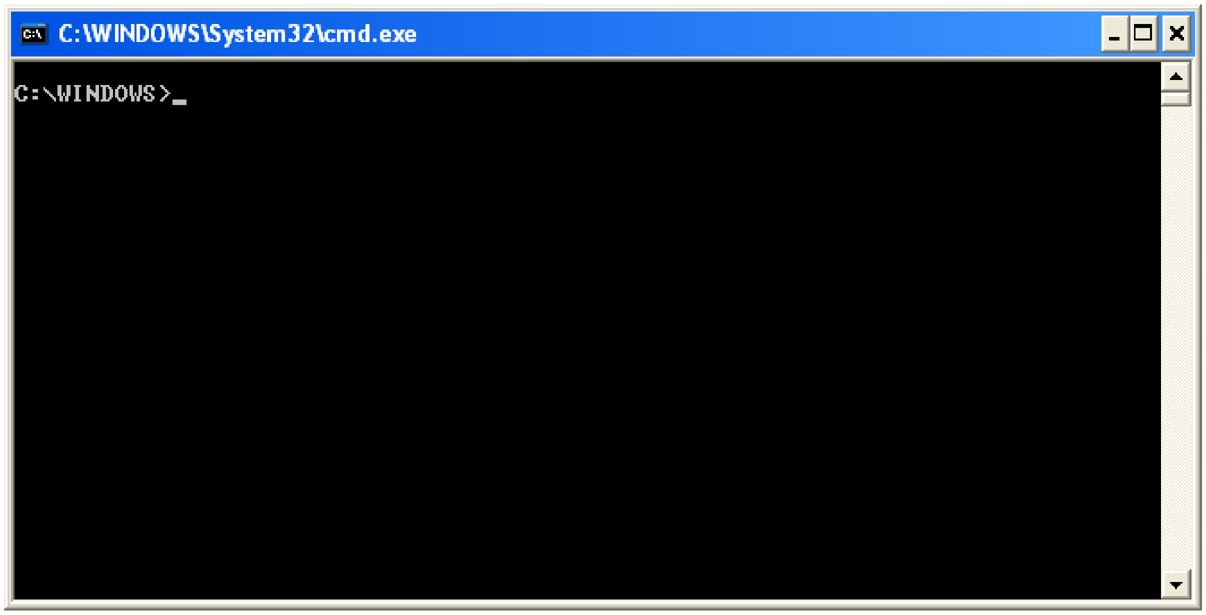
Run a program on Windows

To run a program on Windows, you would do the following steps:

- Click on the Start menu
- Click on the Run option
- Type the name and the directory of the file in the Open field (or click on Browse button if you do not know its location)
- Click on the OK button

The Command Prompt

Before Windows was created, the most common operating system that runs on PC was DOS. Though Windows does not run on DOS, they do have something called the command prompt, which has a similar appearance to DOS.



```
C:\WINDOWS>_
```

To use the command prompt you would type in the commands and instructions you want and press enter.

The Recycle Bin

The recycle bin provides a safety net when deleting files and folders. When you delete any of these items from your hard disk, Windows places it in the Recycle Bin. Items deleted from a floppy disk or a network drive are permanently deleted and are not sent to the Recycle Bin.

Items in the Recycle Bin remain there until you decide to permanently delete them from your computer.

To delete or restore files in the Recycle Bin

On the desktop, double-click Recycle Bin.Do one of the following:

- To restore an item, right-click it, and then click Restore.
- To restore all of the items, on the Edit menu, click Select All, and then on the File menu, click Restore.
- To delete an item, right-click it, and then click Delete.

Computer Networks

History of Computer Networks

Computer network

Computer network is composed of multiple connected computers that communicate over a wired or wireless medium to share data and other resources.

Network data protocols are used to communicate on the network between computers.

The size and scalability of any computer network are determined both by the physical medium of communication and by the software controlling the communication (i.e., the protocols).

The field of computer networking and today's Internet trace their beginnings back to the early 1960s, a time at which the telephone network was the world's dominant communication network. The global Internet's origin was the Advanced Research Projects Agency Network (ARPANET) of the U.S. Department of Defense in 1969 Nowadays, computer networks are developed rapidly

Classification of Computer Networks

Networks can be categorized in several different ways, for example,

- By network layer
- By scale
- By connection method
- By functional relationship
- By network topology
- By protocol

Classification by scale

A Local Area Network (LAN) is a group of computers and associated devices that share a common communications line and typically share the resources of a single processor or server within a small geographic area

A Wide Area Network (WAN) is a computer network that spans a relatively large geographical area (diameter of about 200 km)

GAN (Global Area Network) A network spanning a between geographically distinct cities

Classification by functional relationship

Server based (client/server): Computers set up to be primary providers of services such as file service or mail service.

The computers providing the service are called servers

The computers that request and use the service are called client computers.

Peer-to-peer

Various computers on the network can act both as clients and servers.

Example Many Microsoft Windows based computers allow file and print sharing.

Many networks are combination peer-to-peer and server based networks.

Major Components of a Computer Network

A computer network possibly includes :

- Computers: critical elements of any computer network. They can be considered nodes

- A network card, NIC (network interface card) is a piece of computer hardware designed to allow computers to communicate over a computer network.
- Network media (sometimes referred to as networked media) refers to media mainly used in computer networks : cable, telephone line or wireless.
- Network connection equipments : HUB, SWITCH, ROUTER,etc.
- Network Operating System (NOS) is an operating system that includes special functions for connecting computers and devices into a local-area network (LAN) or Inter-networking. Some popular NOSs for DOS and Windows systems include Novell NetWare, Windows NT and 2000, Sun Solaris and IBM OS/2.
- Network software.
- Network services, for example email.

Network Topology

Network topology is the arrangement or mapping of the elements (links, nodes, etc.) of a network, especially the physical (real) and logical (virtual) interconnections between nodes .

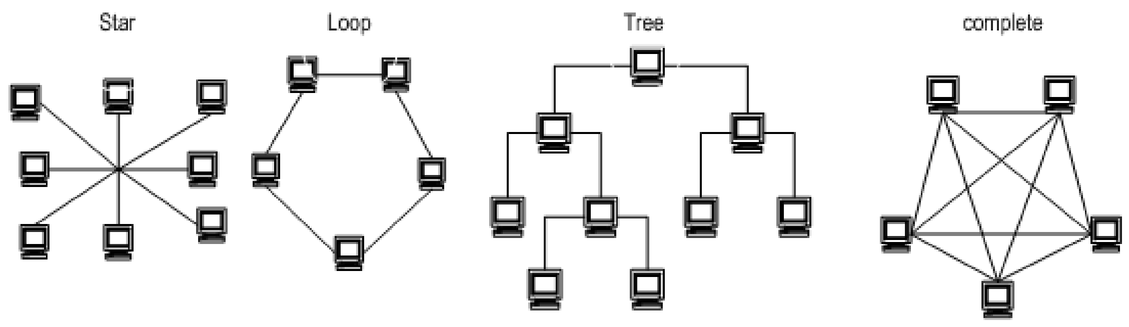
There are three basic categories of network topologies:

- Physical topologies
- Signal topologies
- Logical topologies

Here are some physical topology

Point to point

The simplest topology is a permanent link between two endpoints. Switched point-to-point topologies are the basic model of conventional telephony. The value of a permanent point-to-point network is the value of guaranteed, or nearly so, communications between the two endpoints. The value of an on-demand point-to-point connection is proportional to the number of potential pairs of subscribers.



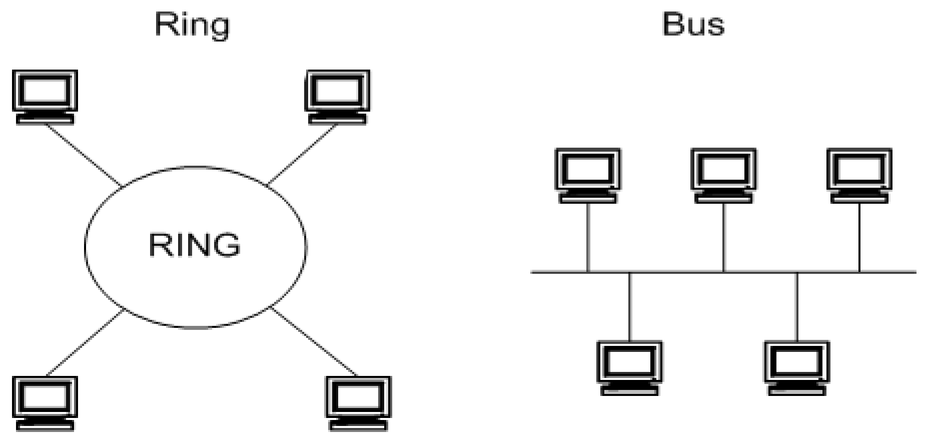
Network Topology (Point to Point)

Bus

Bus networks (not to be confused with the system bus of a computer) use a common backbone to connect all devices. A single cable, the backbone functions as a shared communication medium that devices attach or tap into with an interface connector. A device wanting to communicate with another device on the network sends a broadcast message onto the wire that all other devices see, but only the intended recipient actually accepts and processes the message.

Ring

In a ring network, every device has exactly two neighbors for communication purposes. All messages travel through a ring in the same direction (either "clockwise" or "counterclockwise"). A failure in any cable or device breaks the loop and can take down the entire network.



Network Topology (Broadcast)

The Internet

History of the Internet

The history of the Internet dates back to the early development of communication networks. The idea of a computer network intended to allow general communication between users of various computers has developed through a large number of stages. The melting pot of developments brought together the network of networks that we know as the Internet. This included both technological developments, as well as the merging together of existing network infrastructure and telecommunication systems.

The earliest versions of these ideas appeared in the late 1950s. Practical implementations of the concepts began during the late 1960s and 1970s. By the 1980s, technologies we would now recognize as the basis of the modern Internet began to spread over the globe. In the 1990s the introduction of the World Wide Web saw its use become commonplace.

Internet Services

- FTP (Filer Transfer Protocol)
- Telnet
- WWW
- Email
- Chat

Advantages of the Internet

The Internet or the World Wide Web is indeed a wonderful and amazing addition in our lives. The Internet can be known as a kind of global meeting place where people from all parts of the world can come together. The major advantages of the internet are:

- E-mail: E-mail is an online correspondence system. With e-mail you can send and receive instant electronic messages, which works like writing letters.
- Access Information: The Internet is a virtual treasure trove of information. The ‘search engines’ on the Internet can help you to find data on any subject that you need.
- Shopping: Along with getting information on the Internet, you can also shop online. There are many online stores and sites that can be used to look for products as well as buy them using your credit card
- Online Chat: There are many ‘chat rooms’ on the web that can be accessed to meet new people, make new friends, as well as to stay in touch with old friends.
- Downloading Software: You can download innumerable, games, music, videos, movies, and a host of other entertainment software from the Internet, most of which are free.

How to Connect to the Internet?

Before you can connect to the Internet and access the World Wide Web, you need to have certain equipment:

- The Hardware: Modem (dial up, ADSL) or Ethernet Card
- The Software: Operating System, Connection Software
- The Browser
- Connection Options: Dial up, Cable, ADSL, Wireless. . .
- Locating Internet Access Providers

Introduction to C

History of the C Programming Language

C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators.

C was developed at Bell Laboratories in 1972 by Dennis Ritchie. Many of its principles and ideas were taken from the earlier language B and B's earlier ancestors BCPL and CPL. CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both high level, machine independent programming and would still allow the programmer to control the behavior of individual bits of information.

There are some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

C's power and flexibility soon became apparent. Because of this, the Unix operating system which was originally written in assembly language, was almost immediately re-written in C (only the assembly language code needed to "bootstrap" the C code was kept). During the rest of the 1970's, C spread throughout many colleges and universities because of it's close ties to Unix and the availability of C compilers. Soon, many different organizations began using their own versions of C causing compatibility problems. In response to this in 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C which became known as ANSI Standard C. Today C is in widespread use with a rich standard library of functions.

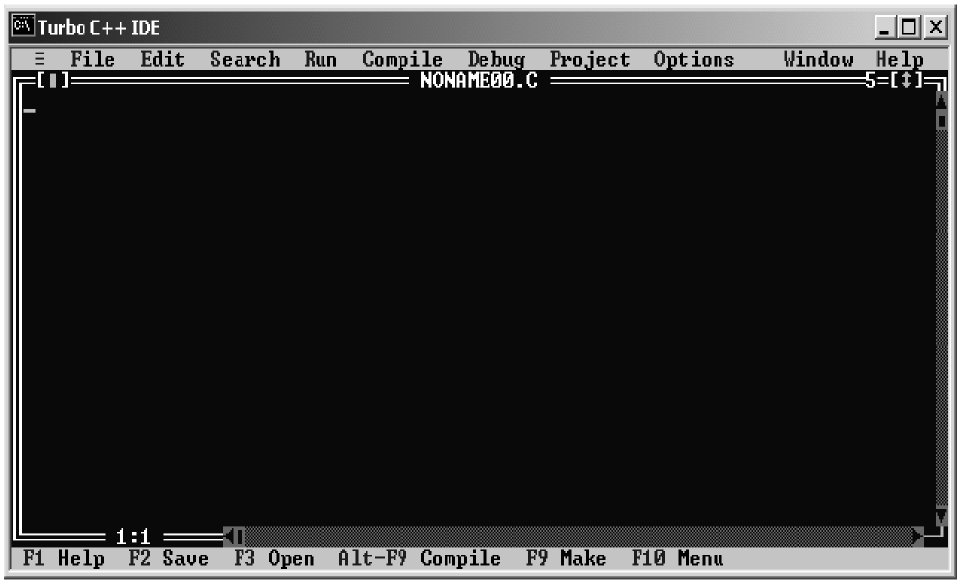
The Integrated Development Environment of C++ 3.0

Start Turbo C++ IDE

Change to directory C:\TC\BIN

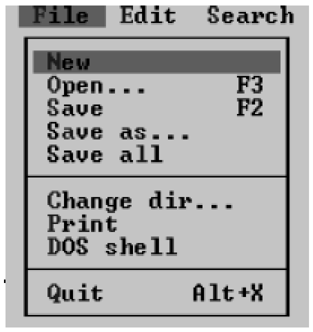
Run TC.EXE

The screen should look like [link]

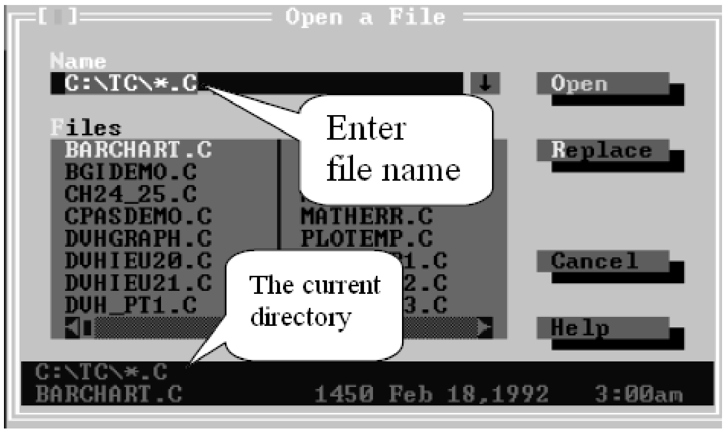

Turbo C++ IDE
≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME00.C
1:1
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

C++ IDE

Select the File menu to create a new file or open an existing file to edit.



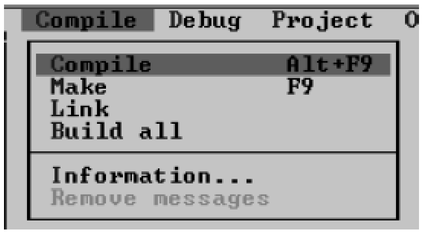
Choose a file to open. This screen allow you to change to your directory.



Change directory dialog box

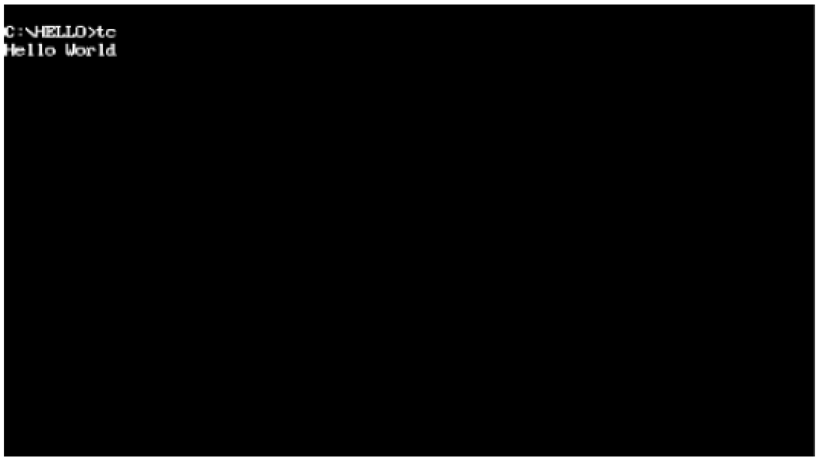
Compile and run a program

To compile a program , use F9 or the compile menu



To run a program, use the run menu or press Ctrl + F9

If your program have no syntax error, the user screen look like in [link]



```
C:\HELLO>tc
Hello World
```

The user screen of C++ 3.0

To exit from IDE, select the file menu, choose quit or press Alt+X

Basic Components of C Programs

Symbols

A C program consists of the following characters:

26 capital letter of English alphabet : A, B, C, D, ... X, Y, Z

26 small letter of English alphabet : a, b, c, d, ...x, y, z

10 digits : 0, 1, . . . 9

Math operators : + - * / = < >

Other symbols : |, \, # , %, ~, ...

Key Words

A keyword is an identifier which indicate a specific command. Keywords are also considered reserved words. You shouldn't use them for any other purpose in a C program.

The most important keywords of Turbo C are

asm	auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for	goto
if	int	long	register	return	short	signed	sizeof
static	struct	switch	typedef	union	unsigned	void	volatile
while							

Identifiers

Identifiers or names refer to a variety of things : functions; tag of structures, union and enumerations; member of structures or unions; enumeration constants; typedef names and objects. There are some restrictions on the names .

Names are made up of letters and digit; The first character must be a letter. The underscore "_" count as a letter; sometime it is useful for improving the readability of long variable names. For example, name **unit_price** is easier to understand than **unitprice**. However, don't begin variable names with underscore, since library routines often use such names.

Upper and lower case are distinct, so x and X are different names. Traditional C practice use lower case for variable names, and all upper case for symbolic constant.

Only the first 31 characters are significant.

Keywords are reserved: you can't use them as variable names.

Example The following names are valid

i, x, b55, max_val

and the following names are invalid

12w	the first character is a digit
income tax	use invalid character " "
char	char is a keyword

It is wise to choose variable names that are related to the purpose of the variable, for example, count_of_girls, MAXWORD.

Data Types

Data is valuable resources of computers. Data may comprise numbers, text, images . . .

They belong to different data types.

In programming languages, a data type is a set of values and the operations on those values.

For example, int type is the set of 32-bit integers within the range -2,147,483,648 to 2,147,483,647 together with the operations described in the following table.

Operations	Symbol
Opposite	-
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Equal to	= =
Greater than	>
Less than	<
…	

A data type can also be thought of as a constraint placed upon the interpretation of data in a type system in computer programming.

Common types of data in programming languages include primitive types (such as integers, floating point numbers or characters), tuples, records, algebraic data types, abstract data types, reference types, classes and function types. A data type describes representation, interpretation and structure of values manipulated by algorithms or objects stored in computer memory or other storage device. The type system uses data type information to check correctness of computer programs that access or manipulate the

data.

Constants

In general, a constant is a specific quantity that does not or cannot change or vary. A constant's value is fixed at compile-time and cannot change during program execution. C supports three types of constants : numeric, character, string.

Numeric constants of C are usually just the written version of numbers. For example 1, 0, 56.78, 12.3e-4. We can specify our constant in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero : -0.15
- Hexadecimal constants are written with a leading 0x : 0x1ae
- Long constants are written with a trailing L : 890L or 890l

Character constants are usually just the character enclosed in single quotes; ‘a’, ‘b’, ‘c’. Some characters can’t be represented in this way, so we use a 2 character sequence (escape sequence).

‘\n’	newline
‘\t’	horizontal tab
‘\v’	vertical tab
‘\b’	backspace
‘\r’	carriage return
‘\’	backslash
‘\’’	single quote
‘\”’	double quotes
‘\0’	null (used automatically to terminate character strings)

Character constants participate in numeric operations just as any other integers (they are represented by their order in the ASCII character set), although they are most often used in comparison with other characters.

Character constants are rarely used, since string constants are more convenient. A string constant is a sequence of characters surrounded by double quotes e.g. “Brian and Dennis”.

A character is a different type to a single character string. This is important.

It is helpful to assign a descriptive name to a value that does not change later in the program. That is the value associated with the name is constant rather than variable, and thus such a name is referred to as symbolic constant or simply a constant.

Variables

Variables are the names that refer to sections of memory into which data can be stored.

Let’s imagine that memory is a series of different size boxes. The box size is memory storage area required in bytes.In order to use a box to store data, the box must be given a name, this process is known as **declaration**. It helps if you give a box a meaningful name that relates to the type of information and it is easier to find the data.The boxes must be of the correct size for the data type you are going to put into it. An integer number such as 2 requires a smaller box than a floating point number as 123e12.

Data is placed into a box by assigning the data to the box. By using the name of the box you can retrieve the box contents, some kind of data.

Variable named by an identifier. The conventions of identifiers were shown in 1.3.3.

Names should be meaningful or descriptive, for example, studentAge or student_age is more meaningful than age, and much more meaniful than a single letter such as a.

Operators

Programming languages have a set of operators that perform arithmetical operations , and others such as Boolean operations on truth values, and string operators manipulating strings of text. Computers are mathematical devices , but compilers and interpreters require a full syntactic theory of all operation in order to parse formulae involving any combination correctly.

Expressions

An expression in a programming language is a combination of values, functions, etc. interpreted according to the particular rules of precedence and association for a particular programming language, which computes and returns another value.

C expressions are arranged in the following groups based on the operators they contain and how you use them:

- Arithmetic expression
- Conditional expression
- Assignment expression
- Comma expression
- lvalue
- Constant expression

Expressions are used as

- Right hands of assignment statements
- Actual parameters of functions
- Conditions of if statements
- Indexes of while statements
- Operands of other expressions . . . .

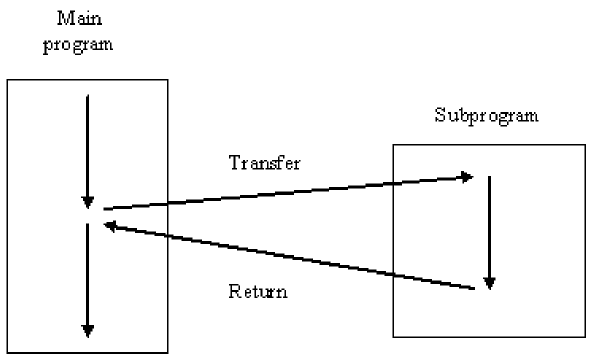
Functions

A subprogram (also known as a procedure or subroutine) is nothing more than a collection of instructions forming a program unit written independently of the main program yet associated with it through a transfer/return process. Control is passed to the subprogram at the time its services are required, and then control is returned to the main program after the subprogram has finished.

The syntax used to represent the request of subprogram varies among the different language. The techniques used to describe a subprogram also varies from language to language. Many systems allow such program units to be written in languages other than that of the main program.

In most procedural programming languages, a subprogram is implemented as though it were completely separate entity with its own data and algorithm so that an item of data in either the main program or the subprogram is not automatically accessible from within the other. With this arrangement, any transfer of data between the two program parts must be specified by the programmer. This is usually done by listing the items called parameters to be transferred in the same syntactic structure used to request the subprogram's execution.

The names used for the parameters within the subprogram can be thought of as merely standing in for the actual data values that are supplied when the subprogram is requested. As a result, you often hear them called formal parameters, whereas the data values supplied from the main program are refereed to actual parameters.



C only accept one kind of subprogram, function. A function is a sub program in which input values are transferred through a parameter list. However, information is returned from a function to the main program in the form of the "value of the function". That is the value returned by a function is associated with the name of the function in a manner similar to the association between a value and a variable name. The difference is that the value associated with a function name is computed (according to the function's definition) each time it is required, whereas when a variable 's value is required, it is merely retrieve from memory.

C also provide a rich collection of built-in functions.There are more than twenty functions declared in <math.h>. Here are some of the more frequently used.

Name	Description	Math Symbols	Example
sqrt(x)	square root	$\sqrt{x}$	sqrt(16.0) is 4.0
pow(x,y)	compute a value taken to an exponent, xy	x^y	pow(2,3) is 8
exp(x)	exponential function, computes ex	e^y	exp(1.0) is 2.718282
log(x)	natural logarithm	ln x	log(2.718282) is 1.0
log10(x)	base-10 logarithm	log x	log10(100) is 2
sin(x)	sine	sin x	sin(0.0) is 0.0
cos(x)	cosine	cos x	cos(0.0) is 1.0
tan(x)	tangent	tg x	tan(0.0) is 0.0
ceil(x)	smallest integer not less than parameter	$\lceil x \rceil$	ceil(2.5) is 3; ceil(-2.5) is –2
floor(x)	largest integer not greater than parameter	$\lfloor x \rfloor$	floor(2.5) is 2; floor(-2.5) is –3

Library

The library is not part of the C language proper, but an environment that support C will provide the function declarations and type and macro definitions of this library.The functions, types and macro of the library are declared in headers.

C header files have extensions .h. Header files should not contain any source code. They are used purely to store function prototypes, common #define constants, and any other information you wish to export from the C file that the header file belongs to.

A header can be accessed by

```
 #include <header>
```

Here are some headers of Turbo C library

stdio.h Provides functions for performing input and output.

stdlib.h Defines several general operation functions and macros.

conio.h Declares several useful library functions for performing "console input and output" from a program.

math.h Defines several mathematic functions.

string.h Provides many functions useful for manipulating strings (character arrays).

io.h Defines the file handle and low-level input and output functions

graphics.h Includes graphics functions

Statements

A statement specifies one or more action to be perform during the execution of a program.

C requires a semicolon at the end of every statement.

Comments

Comments are marked by symbol “/*” and “*/”. C also use // to mark the start of a comment and the end of a line to indicate the end of a comment.

Example:

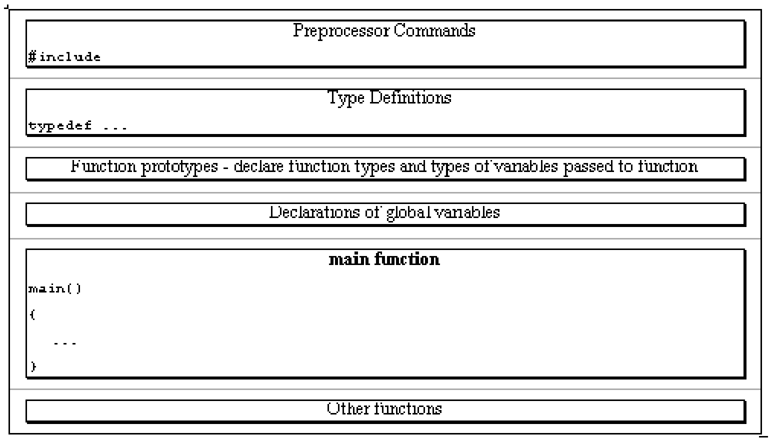
```
1. The Hello program written using the first  commenting style of
C
/* A simple program to demonstrate
  C style comments

  The following line is essential
  in the C version of the hello HUT program
  */
#include <stdio.h>
main()
{
        printf  /* just print */  ("Hello HUT\n");

}

The Hello program written using the second commenting style of C

//  A simple program to demonstrate
//  C style comments
//
//  The following line is essential
//  in the C version of the hello HUT program
#include <stdio.h>
main()
{
        printf("Hello HUT\n"); //print the string and then go to a
new line

}
```

By the first way, a program may have a multi-line comments and comments in the middle of a line of code.However, you shouldn't mix the two style in the same program.

C program structure

A C program basically has the following form:

- Preprocessor Commands : Declare standard libraries used inside the program
- Type definitions : Define new data types used inside the program
- Function prototypes : Declare function types and variables passed to function.
- Variables : State the names and data types of global variables
- Functions: Include the main function(required) and the functions that their prototypes were announced above.


Preprocessor Commands
#include
Type Definitions
typedef ...
Function prototypes - declare function types and types of variables passed to function
Declarations of global variables
main function
main()
{
...
}
Other functions

Data Types and Expressions

Standard Data Types

The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. Standard data types in C are listed in the following table:

Variable Type	**Keyword**	**Range**	**Storage in Bytes**
Character	char	-127 to 127	1
Unsigned character	unsigned char	0 to 255	1
(Signed) integer	int	-32,768 to 32,767	2
Unsigned integer	unsigned int	0 to 65,535	2
Short integer	short	-32,768 to 32,767	2
Unsigned short integer	unsigned short	0 to 65,535	2

Variable Type	**Keyword**	**Range**	**Storage in Bytes**
Long integer	long	-2,147,483,648 to 2,147,483,647	4
Unsigned long	integerunsigned long	0 to 4,294,967,295	4
Single precision floating point	float	1.2E-38 to 3.4E38, approx. range precision = 7 digits.	4
Double precision floating point	double	2.2E-308 to 1.8E308, approx. range precision = 19 digits.	8

Table of Variable types

Declaration and Usage of Variables and Constants

Variables

A variable is an object of a specified type whose value can be changed. In programming languages, a variable is allocated a storage location that can contain data that can be modified during program execution. Each variable has a name that uniquely identifies it within its level of scope.

In C, a variable must be **declared before use**, although certain declarations can be made implicitly by content. Variables can be declared at the start of any block of code, but most are found at the start of each function. Most local variables are created when the function is called, and are destroyed on return from that function.

A declaration begins with the type, followed by the name of one or more variables. Syntax of declare statement is described as:

```
data_type  list_of_variables;
```

A list of variables includes one or many variable names separated by commas.

Example:
Single declarations

```
int age;         //integer variable
float amountOfMoney;//float variable
char initial;// character variable
```

Multiple declarations:

```
int age, houseNumber, quantity;
float distance, rateOfDiscount;
char firstInitial, secondInitial;
```

Variables can also be initialized when they are declared, this is done by adding an equals sign and the required value after the declaration.

Example:

```
int high = 250;     //Maximum Temperature
int low = -40;      //Minimum Temperature
int results[20];    //Series of temperature
readings
```

Constants

A constant is an object whose value cannot be changed. There are two method to define constant in C:

- By #define statement. Syntax of that statement is:

```
#define  constant_name value
```

Example:

```
#define  MAX_SALARY_LEVEL  15 //An integer
constant
#define  DEP_NAME  “Computer Science”
// A string constant
```

- By using const keyword

```
const data_type  variable_name = value;
```

Example:

```
const double e = 2.71828182845905;
```

Functions printf, scanf

Usually i/o, input and output, form the most important part of any program. To do anything useful your program needs to be able to accept input data and report back your results. In C, the standard library (stdio.h) provides routines for input and output. The standard library has functions for i/o that handle input, output, and character and string manipulation. Standard input is usually means input using the keyboard. Standard output is usually means output onto the monitor.

- Input by using the scanf() function
- Output by using the printf() function

To use printf and scanf functions, it is required to declare the header <stdio.h>

The printf() function

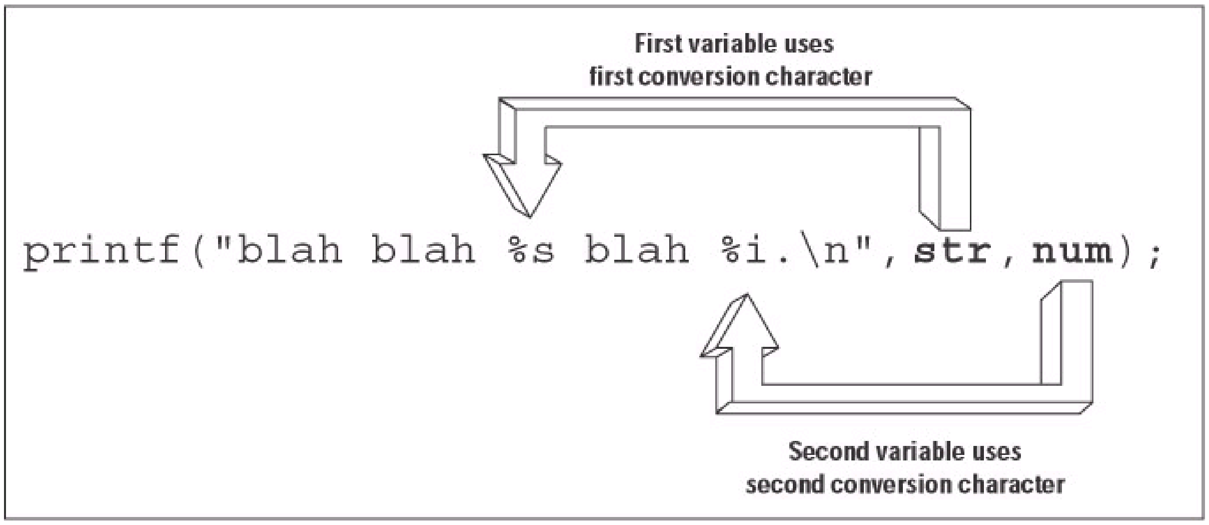
The standard library function printf is used for formatted output. It makes the user input a string and an optional list of variables or strings to output. The variables and strings are output according to the specifications in the printf() function. Here is the general syntax of printf .

```
printf(“[string]”[,list of arguments]);
```

The **list of arguments** allow expressions, separated by commas.

The **string** is all-important because it specifies the type of each variable in the list and how you want it printed. The string is usually called the control string or the format string. The way that this works is that printf scans the string from left to right and prints on the screen any characters it encounters - except when it reaches a % character.

The % character is a signal that what follows it is a specification for how the next variable in the list of variables should be printed. printf uses this information to convert and format the value that was passed to the function by the variable and then moves on to process the rest of the control string and anymore variables it might specify.


First variable uses
first conversion character
printf("blah blah %s blah %i.\n", str, num);
Second variable uses
second conversion character

For Example

```
printf("Hello World");
```

only has a control string and, as this contains no % characters it results in Hello World being displayed and doesn't need to display any variable values. The specifier %d means convert the next value to a signed decimal integer and so:

```
printf("Total = %d",total);
```

will print Total = and then the value passed by total as a decimal integer.

The % Format Specifiers

The % specifiers that you can use are:

	Usual variable type	**Display**
%c	char	single character

%d (%i)	int	signed integer
%e (%E)	float or double	exponential format
%f	float or double	signed decimal
%g (%G)	float or double	use %f or %e as required
%o	int	unsigned octal value
%s	array of char	sequence of characters
%u	int	unsigned decimal
%x (%X)	int	unsigned hex value

Formatting Your Output

The type conversion specifier only does what you ask of it - it convert a given bit pattern into a sequence of characters that a human can read. If you want to format the characters then you need to know a little more about the printf function's control string.

Each specifier can be preceded by a modifier which determines how the value will be printed. The most general modifier is of the form:

```
flag width.precision
```

The flag can be any of

flag	**meaning**

flag	meaning
-	left justify
+	always display sign
space	display space if there is no sign
0	pad with leading zeros
#	use alternate form of specifier

The width specifies the number of characters used in total to display the value and precision indicates the number of characters used after the decimal point.

For example,

%10.3f will display the float using ten characters with three digits after the decimal point. Notice that the ten characters includes the decimal point, and a - sign if there is one. If the value needs more space than the width specifies then the additional space is used - width specifies the smallest space that will be used to display the value.

%-10d will display an int left justified in a ten character space.

The specifier %+5d will display an int using the next five character locations and will add a + or - sign to the value.

Strings will be discussed later but for now remember: if you print a string using the %s specifier then all of the characters stored in the array up to the first null will be printed. If you use a width specifier then the string will be right justified within the space. If you include a precision specifier then only that number of characters will be printed.

For Example

```
printf("%s,Hello")
```

will print Hello,

```
printf("%25s ,Hello")
```

will print 25 characters with Hello right justified

Also notice that it is fine to pass a constant value to printf as in printf("%s,Hello").

Finally there are the control codes:

\b	backspace
\f	formfeed
\n	new line
\r	carriage return
\t	horizontal tab
\'	single quote
\0	null

If you include any of these in the control string then the corresponding ASCII control code is sent to the screen, or output device, which should produce the effect listed. In most cases you only need to remember \n for new line.

The scanf() function

The scanf function works in much the same way as the printf. That is it has the general form:

```
scanf("control string",variable,variable,...)
```

In this case the control string specifies how strings of characters, usually typed on the keyboard, should be converted into values and stored in the listed variables. However there are a number of important differences as well as similarities between scanf and printf.

The most obvious is that scanf has to change the values stored in the parts of computers memory that is associated with parameters (variables).

To understand this fully you will have to wait until we have covered functions in more detail. But, just for now, bare with us when we say to do this the scanf function has to have the addresses of the variables rather than just their values. This means that simple variables have to be passed with a preceding &.

The second difference is that the control string has some extra items to cope with the problems of reading data in. However, all of the conversion specifiers listed in connection with printf can be used with scanf.

The rule is that scanf processes the control string from left to right and each time it reaches a specifier it tries to interpret what has been typed as a value. If you input multiple values then these are assumed to be separated by white space - i.e. spaces, newline or tabs. This means you can type:

3 4 5

or

3

4

5

and it doesn't matter how many spaces are included between items. For Example

```
scanf("%d %d",&i,&j);
```

will read in two integer values into i and j. The integer values can be typed on the same line or on different lines as long as there is at least one white space character between them.

The only exception to this rule is the %c specifier which always reads in the next character typed no matter what it is. You can also use a width modifier in scanf. In this case its effect is to limit the number of characters accepted to the width.

For Example

```
scanf("%l0d",&i)
```

would use at most the first ten digits typed as the new value for i.

Here is an example to demonstrate the usage of printf and scanf functions

```
#include <conio.h>
#include <stdio.h>
void main()
{
// variable declaration
int a;
float x;
char ch;
char* str;
// Enter data
printf(“Input an integer”);
scanf(“%d”,&a);
printf(“\n  Input a real number”);
scanf(“%f”,&x);
printf(“\n Input a character”);
fflush(stdin); scanf(“%c”,&ch);
printf("\n Input a string" );
fflush(stdin); scanf("%s",str);
// Output the data
printf("\n Your data");
printf("\n Integer: %d",a);
printf("\n Float : %.2f",x);
printf("\n Char: %c:,ch);
printf("\n String : %s",str);
}
```

(Function fflush are used to avoid stopping the reading process when meet one or more spaces)

There is one main problem with scanf function which can make it unreliable in certain cases. The reason being is that scanf tends to ignore white spaces, i.e. the space character. If you require your input to contain spaces this can cause a problem.

Scanf will skip over white space such as blanks, tabs and new lines in the input stream. The exception is when trying to read single characters with the conversion specifiers %c. In this case, white space is read in. So, it is more difficult to use scanf for single characters. An alternate technique, using getchar, will be described later.

Other Input and Output Functions

getch

The computer asks the user press a key. This key does not appear on screen. The syntax is:

```
getch();
```

getch() is used to terminate a program when the user press a key.

gets

gets reads a line of input into a character array. The syntax is:

```
gets(name_of_string);
```

puts

puts writes a line of output to standard output. The syntax is:

```
puts(name of string);
```

It terminates the line with a new line, '\n'. It will return EOF is an error occurred. It will return a positive number on success.

For using the statements mentioned above, you must declare the library <conio.h>.

Expressions

Operators

C contains the following operator groups.

- Arithmetic
- Assignment
- Logical/relational
- Bitwise

Arithmetic Operators

The arithmetic operators are +, -, /, * and the modulus operator %. Integer division truncates any fractional part. The expression x%y produces the remainder when x is divided by y, and thus is zero when y divide x exactly.The % operator cannot be applied to a float or double.

The binary + and – operators have the same precedence, which is lower than the precedence of *, / and %, which is turn lower than unary + and - .

Arithmetic operators associate left to right.

Operator	**Meaning**	**Data type of the operands**	**Examples**
-	opposite	integer, float	`int a, b;` `-12; -a;` `-25.6;`
+	addition	integer, float	`float x, y;` `5 + 8; a + x;` `3.6 + 2.9;`
-	subtraction	integer, float	`3 – 1.6; a – 5;`
*	multiplication	integer, float	`a * b; b * y;` `2.6 * 1.7;`
/	division	integer, float	`10.0/3.0;` `(= 3.33…)`

			`10/3.0;`
			`(= 3.33…)`
			`10.0/3;`
			`(= 3.33…)`

/	integer division	integer	`10/3;`
			`(= 3)`

%	modulus	integer	`10%3;`
			`(=1)`

Assignment Operators

These all perform an arithmetic operation on the lvalue and assign the result to the lvalue. So what does this mean in English? Here is an Example

```
counter = counter + 1;
```

can be reduced to

```
counter += 1;
```

Here is the full set.

=	

*=	Multyply
/=	Divide
%=	Modulus
+=	Add
-=	Subtract
<<=	Left Shift
>>=	Right Shift
&=	Bitwise AND
^=	Bitwise Exclusive OR (XOR)
|=	Bitwise Ixclusive OR

if expr1 and expr2 are expressions then

expr1 op= expr2 is equivalent to expr1 = expr1 op expr2

Logical and Relational Operators

Relational operators

Operators	**Meaning**	**Examples**
>	greater than	

		`2 > 3 (is 0)`
		`6 > 4 (is 1)`
		`a > b`

>=	greater than or equal to	`6 >= 4 (is 1)`
		`x >= a`

<	less than	`5 < 3 (is 0),`

<=	less than or equal to	`5 <= 5 (is 1)`
		`2 <= 9 (is 1)`

==	equal to	`3 == 4 (is 0)`
		`a == b`

!=	not equal to	`5 != 6 (is 1)`
		`6 != 6 (is 0)`

Logical operators

Operators	Meaning	Data types of the operands	Examples
&&	logical and	2 logic expressions	`3<5 && 4<6 (is 1)` `2<1 && 2<3 (is 0)` `a > b && c < d`
||	logical or	2 logic expressions	`6 || 0 (is 1)` `3<2 || 3<3 (is 1)` `x >= a || x == 0`
!	logical not	1 logic expression	`!3      (is 0)` `!(2>5)  (is 1)`

Bitwise Operators

Operators	Meaning	Data types	Examples

		of the operands	

&	Binary AND	2 binary numbers	`0 & 0`
			`(is 0)`
			`0 & 1`
			`(is 0)`
			`1 & 0`
			`(is 0)`
			`1 & 1`
			`(is 1)`
			`101 & 110`
			`(is 100)`

|	Binary OR	2 binary numbers	`0 | 0`
			`(is 0)`
			`0 | 1`
			`(is 0)`
			`1 | 0`
			`(is 0)`
			`1 | 1`
			`(is 1)`
			`101 | 110`
			`(is 111)`

^	Binary XOR	2 binary numbers	`0 ^ 0`
			`(is 0)`
			`0 ^1`
			`(is 1)`
			`1 ^ 0`

			`(is 1)`
			`1 ^ 1`
			`(is 0)`
			`101 ^ 110`
			`(is 011)`

<<	Shift left	1 binary number	`a << n`
			`(is a*2n)`
			`101 << 2`
			`(is 10100)`

>>	Shift right	1 binary number	`a >> n`
			`(is a/2n)`
			`101 >> 2`
			`(is 1)`

~	One's complement	1 binary number	`~ 0`
			`(is 1)`
			`~ 1`
			`(is 0)`
			`~ 110`
			`(is 001)`

Increment and Decrement Operators

Incrementing, decrementing and doing calculations on a variable is a very common programming task and C has quicker ways of writing the code. The code is rather cyptic in appearance.

The increment operator ++ adds 1 to its operand while the decrement operator - -subtract 1. We have frequently used ++ to increment variables, as in

```
if  (c = = ’\n’)
     ++n;
```

The unusual aspect is that ++ and - - may be used either as prefix operators (before the variable, as in ++n) or postfix operators (after the variable, as in n++). In both cases, the effect is to increment n. But the expression ++n increments n before its value is used, while n++ increment n after its value has been used. This mean that in a context where the value is being used, not just the effect, ++n and n++ are different. For example, if n is 5, then

```
x = n++;
```

sets x to 5 but

```
x = ++n;
```

sets x to 6. In both cases, n becomes 6.

Note: The increment and decrement operator can only be applied to variables; an expression like (i + j)++ is illegal.

Memory Addressing Operators

The five operators listed in [missing_resource: mk:@MSITStore:C:%5CDOCUME~1%5CComputer%5CLOCALS~1%5CTemp%5CRar$DI00.969%5CC.in.a.Nutshell.(OReilly)-0596006977.chm::]

Operator	**Meaning**	**Example**	**Result**
&	Address of	&x	Pointer to x
*	Indirection operator	*p	The object or function that p points to
[ ]	Subscripting	x[y]	The element with the index y in the array x (or the element with the index x in the array y: the [ ] operator works either way)
.	Structure or union member designator	x.y	The member named y in the structure or union x
->	Structure or union member designator by reference	p->y	The member named y in the structure or union that p points to

Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversion era those that convert a narrower operand into a wider one without loosing information, such as converting an integer into floating point .

If there are no unsigned operands, the following informal set of rules will suffice:

If either operand is long double, convert the other to long double.

Otherwise, if either operand is double, convert the other to double.

Otherwise if either operand is float, convert the other to float.

Otherwise convert char and short to int.

Then if either operand is long, convert the other to long.

A char is just a small integer, so chars may be freely used in arithmetic expressions

Precedence of Operators

Operators listed by type.

All operators on the same line have the same precedence. The first line has the highest precedence.

Level	**Operators**	**Associativity**
1	() [] . -> ++ (postfix) – (postfix)	----->
2	! ~ ++ (prefix) -- (prefix) - * & sizeof	<-----
3	* / %	----->
4	+ -	----->

5	<< >>	----->
6	< <= > >=	----->
7	== !=	----->
8	&	----->
9	^	----->
10	|	----->
11	&&	----->
12	||	----->
13	?:	<-----
14	= += -=	<-----

Note: associate left to right

The Control Flow

The control flow of a language specify the order in which operations are performed. Each program includes many statements. Statements are processed one after another in sequence, except where such control statements result in jumps.

Statements and Blocks

An expression such as x = 0 or i++ or printf(. . . .) becomes a statement when it is followed by a semicolon, as in

```
x=0;
i++;
printf(. . . .);
```

In the C language, the semicolon is a statement terminator.

A block also called a compound statement, or compound statement, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

In blocks, declarations and definitions can appear anywhere, mixed in with other code. Note that there is no semicolon after the right brace that ends a block.

Example:

```
{ into I = 0;      /* Declarations  */
  static long a;
  extern long max;

  ++a;            /* Statements    */
  if( a >= max)
  {   . . .   }  /* A nested block */
  . . .
}
```

An expression statement is an expression followed by a semicolon. The syntax is:

```
[expression] ;
```

Example:

```
y = x;        // Assignment
```

The expression—an assignment or function call, for example—is evaluated for its side effects. The type and value of the expression are discarded.

A statement consisting only of a semicolon is called an empty statement, and does not perform any operation. For Example

```
for ( i = 0;  str[i] != '\0'; ++i )
   ;                     // Empty statement
```

If, If else statements

The if statement has two forms:

```
if(expression) statement
```

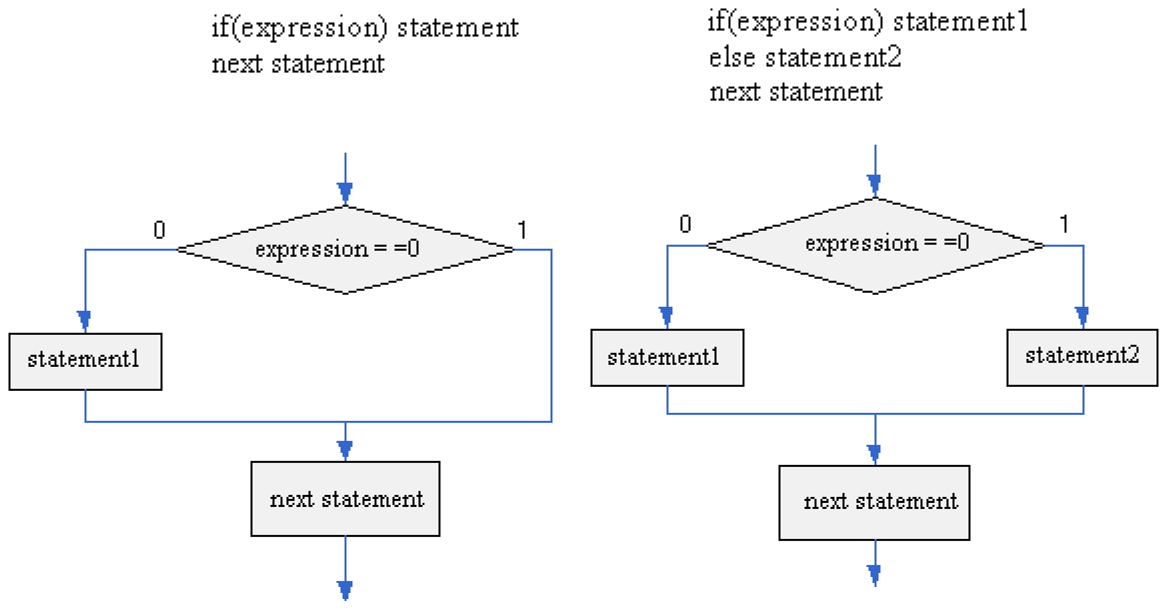
and

```
if(expression) statement1
else statement2
```

In the first form, if (and only if) the expression is non-zero, the statement is executed. If the expression is zero, the statement is ignored. Remember that the statement can be compound; that is the way to put several statements under the control of a single if.

The second form is like the first except that if the statement shown as statement1 is selected then statement2 will not be, and vice versa.

Here are the flowcharts of the two forms of if statement



The form involving else works the same way, so we can also write this.

```
if(expression)
  if(expression)
    statement
  else
    statement
```

this is now ambiguous. It is not clear, except as indicated by the indentation, which of the ifs is responsible for the else. If we follow the rules that the previous example suggests, then the second if is followed by a statement, and is therefore itself a statement, so the else belongs to the first if.

That is not the way that C views it. The rule is that an else belongs to the first if above that hasn't already got an else. In the example we're discussing, the else goes with the second if.

To prevent any unwanted association between an else and an if just above it, the if can be hidden away by using a compound statement, here it is.

```
if(expression){
    if(expression)
            statement
}else
    statement
```

if(expression){

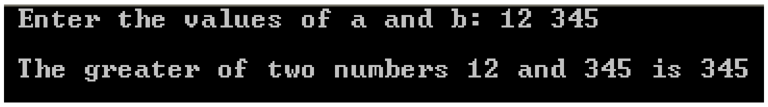
```
if(expression){
    if(expression){
        statement
    }
}else{
    statement
}
```

Example

Example:

```
#include <conio.h>
#include <stdio.h>
void main()
{
```

```
// variable declaration
float a, b;
float max;
printf(" Enter the values of a and b: ");
scanf("%f %f",&a,&b);
if(a<b) //Assign the greater of x and y to the
variable max
        max = b;
else
        max = a;
printf("\n The greater of two numbers %.0f and
%.0f is %.0f ",a,b,max);
getch();
}
```



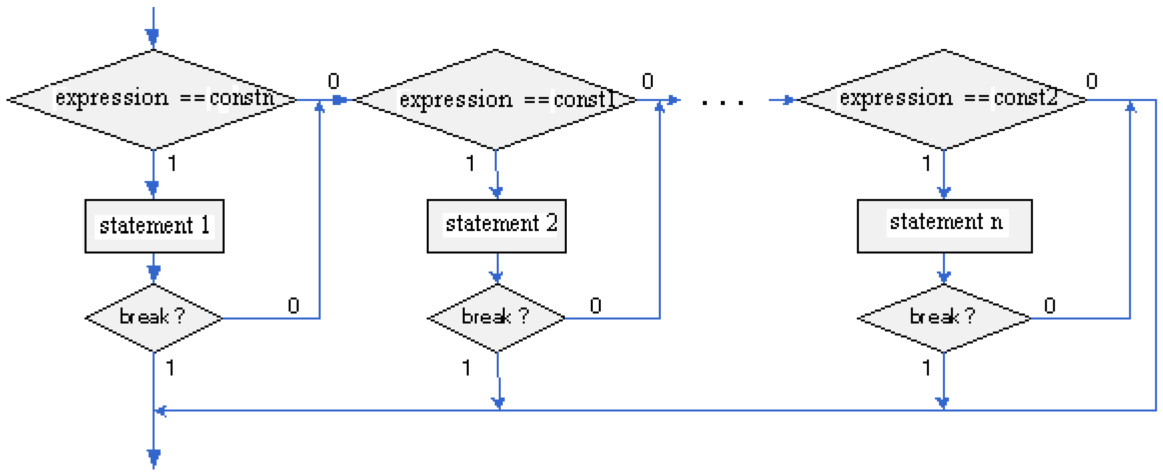
```
Enter the values of a and b: 12 345

The greater of two numbers 12 and 345 is 345
```

The Switch Statement

It is used to select one of a number of alternative actions depending on the value of an expression, and nearly always makes use of another of the lesser statements: the break. It looks like this.

```
switch (expression){
case const1:    statements
case const2:    statements
. . . .
default:        statements
}
```

The flowchart of switch statement is shown below:



The expression is evaluated and its value is compared with all of the const etc. expressions, which must all evaluate to different constant values (strictly they are integral constant expressions). If any of them has the same value as the expression then the statement following the case label is selected for execution. If the default is present, it will be selected when there is no matching value found. If there is no default and no matching value, the entire switch statement will do nothing and execution will continue at the next statement.

Example:

```
OK=1;
switch (OP)
{
        case ‘+’:
z=x+y;
break;
        case ‘-’:
z=x-y;
break;
        case ‘*’:
z=x*y;
break;
case ’/’:
```

```
if (y!=0 )
z=x/y;
                        else OK=0;
        default :
OK=0;
}
```

- The switch requires an integer-compatible value. This value may be a constant, variable, function call, or expression. The switch statement does not work with floating – point data types.
- The value after each case label must be a constant.
- C++ does not support case label with ranges of values. Instead, each value must appear in a separate case label.
- You need to use a break statement after each set of executable statements. The break statement causes program execution to resume after the end of the current switch statement. If you do not use the break statement, the program execution resumes at subsequent case labels.
- The default clause is a catch-all clause.
- The set of statements in each case or grouped case labels need not be enclosed in open and close braces.

The following program writes out the day of the week depending on the value of an integer variable day. It assumes that day 1 is Sunday.

```
#include <stdio.h>
#include <conio.h>
void main()
{int day;
printf(“Enter the value of a weekday”);
scanf(“%d”,&day);
switch (day)
  {
   case 1 : printf( "Sunday");
            break;
   case 2 : printf( "Monday");
            break;
   case 3 : printf( "Tuesday");
            break;
   case 4 : printf("Wednesday");
            break;
   case 5 : printf("Thursday");
            break;
   case 6 : printf("Friday");
            break;
   case 7 : printf("Saturday");
            break;
  default : printf("Not an allowable day number");
            break;
  }
getch();
}
```

If it has already been ensured that day takes a value between 1 and 7 then the default case may be missed out. It is allowable to associate several case labels with one statement. For example if the above example is amended to write out whether day is a weekday or is part of the weekend:

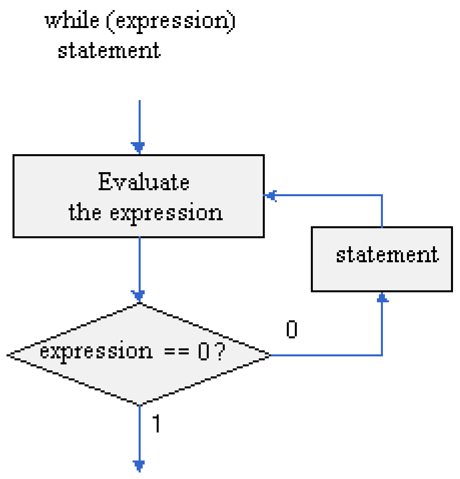
```
switch (day)
  {
    case 1 :
    case 7 : printf( "This is a weekend day");
             break;
    case 2 :
    case 3 :
    case 4 :
    case 5 :
    case 6 : printf( "This is a weekday");
             break;
    default : printf( "Not a legal day");
              break;
}
```

Loops : While and Do While, For

The while statement

The syntax of while statement is simple:

```
while(expression)
    statement
```



The statement is only executed if the expression is non-zero. After every execution of the statement, the expression is evaluated again and the process repeats if it is non-zero. What could be plainer than that? The only point to watch out for is that the statement may never be executed, and that if nothing in the statement affects the value of the expression then the while will either do nothing or loop for ever, depending on the initial value of the expression.

Example:

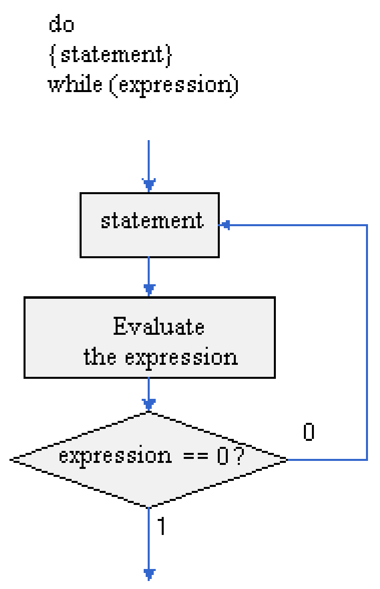
```
#include <stdio.h>
#include <stdlib.h>
```

```
main(){
      int i;

      /* initialize */
      i = 0;
      /* check */
      while(i <= 10){
              printf("%d\n", i);
              /* update */
              i++;
      }
      }
```

The do statement

It is occasionally desirable to guarantee at least one execution of the statement following the while, so an alternative form exists known as the do statement. It looks like this:

```
do
    statement
while(expression);
```



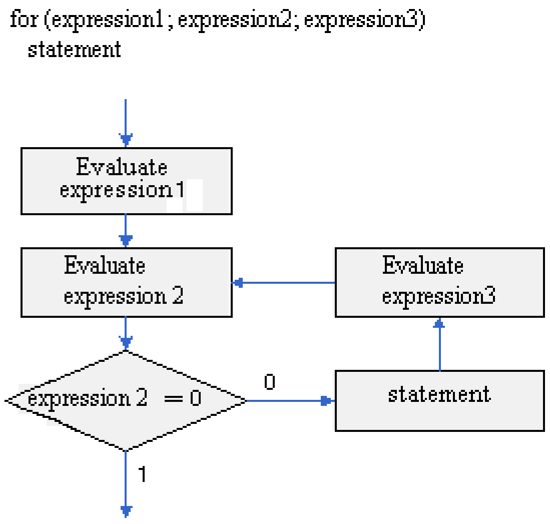
and you should pay close attention to that semicolon—it is not optional! The effect is that the statement part is executed before the controlling expression is evaluated, so this guarantees at least one trip around the loop. It was an unfortunate decision to use the keyword while for both purposes, but it doesn't seem to cause too many problems in practice.

The for statement

A very common feature in programs is loops that are controlled by variables used as a counter. The counter doesn't always have to count consecutive values, but the usual arrangement is for it to be initialized outside the loop, checked every time around the loop to see when to finish and updated each time around the loop. There are three important places, then, where the loop control is concentrated: initialize, check and update. This example shows them.

As you will have noticed, the initialization and check parts of the loop are close together and their location is obvious because of the presence of the while keyword. What is harder to spot is the place where the update occurs, especially if the value of the controlling variable is used within the loop. In that case, which is by far the most common, the update has to be at the very end of the loop: far away from the initialize and check. Readability suffers because it is hard to work out how the loop is going to perform unless you read the whole body of the loop carefully. What is needed is some way of bringing the initialize, check and update parts into one place so that they can be read quickly and conveniently. That is exactly what the for statement is designed to do. Here it is.

```
for (expression1; expression2; expression3)
statement
```



The first expression (expression1) is the initialize part; nearly always an assignment expression which is used to initialize the control variable. After the initialization, the check expression (expression2) is evaluated: if it is non-zero, the statement is executed, followed by evaluation of the update expression (expression3) which generally increments the control variable, then the sequence restarts at the check. The loop terminates as soon as the check evaluates to zero.

There are two important things to realize about that last description:

one, that each of the three parts of the for statement between the parentheses are just expressions;

two, that the description has carefully explained what they are intended to be used for without proscribing alternative uses—that was done deliberately. You can use the expressions to do whatever you like, but at the expense of readability if they aren't used for their intended purpose.

Here is a program that does the same thing twice, the first time using a while loop, the second time with a for. The use of the increment operator is exactly the sort of use that you will see in everyday practice.

Example:

```
#include <stdio.h>
#include <stdlib.h>

void main(){
        int i;
      /* the same done using ``for'' */
      for(i = 0; i <= 10; i++){
              printf("%d\n", i);
      }
      }
```

There isn't any difference between the two, except that in this case the for loop is more convenient and maintainable than the while statement. You should always use the for when it's appropriate; when a loop is being controlled by some sort of counter. The while is more at home when an indeterminate number of cycles of the loop are part of the problem.

Any of the initialize, check and update expressions in the for statement can be omitted, although the semicolons must stay. This can happen if the counter is already initialized, or gets updated in the body of the loop. If the check expression is omitted, it is assumed to result in a ‘true’ value and the loop never terminates. A common way of writing never-ending loops is either

```
for(;;)
```

or

```
while(1)
```

and both can be seen in existing programs.

Loop Flow Control

The control of flow statements that we've just seen are quite adequate to write programs of any degree of complexity. They lie at the core of C and even a quick reading of everyday C programs will illustrate their importance, both in the provision of essential functionality and in the structure that they emphasize. The remaining statements are used to give programmers finer control or to make it easier to deal with exceptional conditions. Only the switch statement is enough of a heavyweight to need no justification for its use; yes, it can be replaced with lots of ifs, but it adds a lot of readability. The others, break, continue and goto, should be treated like the spices in a delicate sauce. Used carefully they can turn something commonplace into a treat, but a heavy hand will drown the flavor of everything else.

The break statement

This is a simple statement. It only makes sense if it occurs in the body of a switch, do, while or for statement. When it is executed the control of flow jumps to the statement immediately following the body of the statement containing the break. Its use is widespread in switch statements, where it is more or less essential to get the control that most people want.

The use of the break within loops is of dubious legitimacy. It has its moments, but is really only justifiable when exceptional circumstances have happened and the loop has to be abandoned. It would be nice if more than one loop could be abandoned with a single break but that isn't how it works. Here is an example

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;

      for(i = 0; i < 10000; i++){
              if(getchar() == 's')
                      break;
              printf("%d\n", i);
      }
      }
```

It reads a single character from the program's input before printing the next in a sequence of numbers. If an ‘s’ is typed, the break causes an exit from the loop.

If you want to exit from more than one level of loop, the break is the wrong thing to use. The goto is the only easy way, but since it can't be mentioned in polite company, we'll leave it till last.

The continue statement

This statement has only a limited number of uses. The rules for its use are the same as for break, with the exception that it doesn't apply to switch statements. Executing a continue starts the next iteration of the smallest enclosing do, while or for statement immediately. The use of continue is largely restricted to the top of loops, where a decision has to be made whether or not to execute the rest of the body of the loop. In this example it ensures that division by zero (which gives undefined behavior) doesn't happen

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;
```

```
        for(i = -10; i < 10; i++){
                if(i == 0)
                        continue;
                printf("%f\n", 15.0/i);
                /*
                 * Lots of other statements .....
                 */
        }
        }
```

Of course the continue can be used in other parts of a loop, too, where it may occasionally help to simplify the logic of the code and improve readability. It deserves to be used sparingly.

Do remember that continue has no special meaning to a switch statement, where break does have. Inside a switch, continue is only valid if there is a loop that encloses the switch, in which case the next iteration of the loop will be started.

There is an important difference between loops written with while and for. In a while, a continue will go immediately to the test of the controlling expression. The same thing in a for will do two things: first the update expression is evaluated, then the controlling expression is evaluated.

Goto and Labels

Everybody knows that the goto statement is a ‘bad thing’. Used without care it is a great way of making programs hard to follow and of obscuring any structure in their flow. Dijkstra wrote a famous paper in 1968 called ‘Goto Statement Considered Harmful’, which everybody refers to and almost nobody has read.

What's especially annoying is that there are times when it is the most appropriate thing to use in the circumstances! In C, it is used to escape from multiple nested loops, or to go to an error handling exit at the end of a function. You will need a label when you use a goto; this example shows both.

```
goto L1;
/* whatever you like here */
L1: /* anything else */
```

A label is an identifier followed by a colon. Labels have their own ‘name space’ so they can't clash with the names of variables or functions. The name space only exists for the function containing the label, so label names can be re-used in different functions. The label can be used before it is declared, too, simply by mentioning it in a goto statement.

Labels must be part of a full statement, even if it's an empty one. This usually only matters when you're trying to put a label at the end of a compound statement—like this.

```
label_at_end: ; /* empty statement */
}
```

The goto works in an obvious way, jumping to the labeled statements. Because the name of the label is only visible inside its own function, you can't jump from one function to another one.

It's hard to give rigid rules about the use of gotos, but, as with the do, continue and the break (except in switch statements), over-use should be avoided. Think carefully every time you feel like using one, and convince yourself that the structure of the program demands it. More than one goto every 3–5 functions is a symptom that should be viewed with deep suspicion.

Pointers and Arrays

From the beginning, we only show how to access or change directly the values of variables through their names. However, the C language provides the developers an effective method to access variables - pointer.

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this Unit also explores this relationship and shows how to exploit it.

Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long.

Any variable in a program is stored at a specific area in memory. If you declare a variable, the compiler will allocate this variable to some consecutive memory cells to hold the value of the variable. The address of the variable is the address of the first memory cell.

One variable always has two properties:

- The address of the variable
- The value of the variable.

Consider the following Example

```
int i, j;
i = 3;
j = i;
```

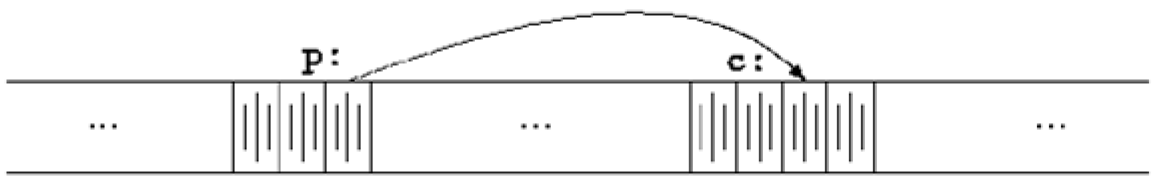
Type of these two variables is integer so they are stored in 2-byte memory area. Suppose that the compiler allocates i at the FFEC address in memory and j in FFEE, we have:

Variable	**Address**	**Value**
i	FFEC	3
j	FFEE	3

Two different variables have different addresses. The i = j assignment affects only on the value of variables, that means the content of the memory area for j will be copied to the content of the memory area for i.

Pointers

A pointer is a group of cells (often two or four) that can hold an address. So if c is a char and p is a pointer that points to it, we could represent the situation this way:



Pointer declaration

If you declare a variable, its name is a direct reference to its value. If you have a pointer to a variable or any other object in memory, you have an indirect reference to its value. A pointer variable stores the address of another object or a function. To start out, the declaration of a pointer to an object that is not an array has the following syntax:

```
type * Name [= initializer];
```

In declarations, the asterisk (*) means "pointer to". The identifier name is declared as an object with the type *, or pointer to type. * is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to.

Here is a simple Example

```
int *iPtr;          // Declare iPtr as a pointer
to int.
double *realptr; // pointer to a double
char *astring; // pointer to a character
```

The type int is the type of object that the pointer iPtr can point to.

To make a pointer refer to a certain object, assign it the address of the object.

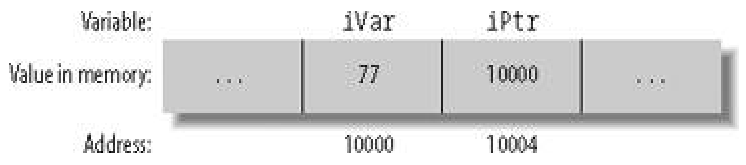
For example, if iVar is an int variable, then the following assignment makes iPtr point to the variable iVar:

```
iPtr = &iVar;       // Let iPtr point to the
variable iVar.
```

In a pointer declaration, the asterisk (*) is part of an individual declarator. We can thus define and initialize the variables iVar and iPtr in one declaration, as follows:

```
int iVar = 77, *iPtr = &iVar; // Define an int
variable and a
                              // pointer to it.
```

The second of these two declarations initializes the pointer iPtr with the address of the variable iVar, so that iPtr points to iVar. Figure 4.1. illustrates one possible arrangement of the variables iVar and iPtr in memory. The addresses shown are purely fictitious examples. As Figure 4.1. shows, the value stored in the pointer iPtr is the address of the object iVar.



A pointer and another object in memory

It is often useful to output addresses for verification and debugging purposes. The **printf()** functions provide a format specifier for pointers: %p. The following statement prints the address and contents of the variable iPtr:

```
printf("Value of iPtr (i.e. the address of iVar):
%p\n"
        "Address of iPtr:
%p\n", iPtr, &iPtr);
```

The size of a pointer in memory given by the expression **sizeof(iPtr)**

& and * operators

The unary operator & gives the address of an object, so the statement

```
p = &c;
```

assigns the address of c to the variable p, and p is said to “point to” c. The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

* is the operator that retrieves the value stored at the address held in the pointer. The indirection operator * yields the location in memory whose address is stored in a pointer. If ptr is a pointer, then *ptr designates the object that ptr points to. Using the indirection operator is sometimes called dereferencing a pointer. The type of the pointer determines the type of object that is assumed to be at that location in memory. For example, when you access a given location using an int pointer, you read or write an object of type int.

The indirection operator * is a unary operator; that is, it has only one operand.ptr points to the variable x. Hence the expression *ptr is equivalent to the variable x itself.

Example:

```
double x, y, *ptr;      // Two double variables and
a pointer to double.
ptr = &x;               // Let ptr point to x.
*ptr = 7.8;             // Assign the value 7.8 to
the variable x.
*ptr *= 2.5;            // Multiply x by 2.5.
y = *ptr + 0.5;       // Assign y the result of the
addition x + 0.5.
```

Do not confuse the asterisk (*) in a pointer declaration with the indirection operator. The syntax of the declaration can be seen as an illustration of how to use the pointer.

```
double *ptr;
```

As declared here, ptr has the type double * (read: "pointer to double"). Hence the expression *ptr would have the type double.

Of course, the indirection operator * must be used with only a pointer that contains a valid address. This usage requires careful programming! Without the assignment ptr = &x in the listing above, all of the statements containing *ptr would be senseless dereferencing an undefined pointer value and might well cause the program to crash.

Pointer Assignment

Since pointers are variables, they can be used without dereferencing. Pointer assignment between two pointers makes them point to the same pointee. So the assignment `iq = ip;` copies the contents of ip into iq, thus making iq point to whatever ip pointed to. It makes iq point to the same pointee as ip. Pointer assignment does not touch the pointees. It just changes one pointer to have the same reference as another pointer. After pointer assignment, the two pointers are said to be "sharing" the pointee.

Example Consider the following programs:

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
}
and
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
```

```
    p1 = p2;
}
```

Suppose the values of variables before executing the last assignment are

Variable	Address	Value
i	FFEC	3
j	FFEE	6
p1	FFDA	FFEC
p2	FFDC	FFEE

After the assignment *p1 = *p2; for the first program:

Variable	Address	Value
i	FFEC	6
j	FFEE	6
p1	FFDA	FFEC
p2	FFDC	FFEE

While the assignment p1 = p2 for the second program results

Variable	Address	Value
i	FFEC	3
j	FFEE	6
p1	FFDA	FFEE
p2	FFDC	FFEE

Initializing Pointers

Pointer variables with automatic storage duration start with an undefined value, unless their declaration contains an explicit initializer. You can initialize a pointer with the following kinds of initializers:

- A null pointer constant.
- A pointer to the same type, or to a less qualified version of the same type.
- A void pointer, if the pointer being initialized is not a function pointer. Here again, the pointer being initialized can be a pointer to a more qualified type.

Operators on Pointers

Besides using assignments to make a pointer refer to a given object or function, you can also modify an object pointer using arithmetic operations.

When you perform **pointer arithmetic**, the compiler automatically adapts the operation to the size of the objects referred to by the pointer type.

You can perform the following operations on pointers to objects:

- Adding an integer to, or subtracting an integer from, a pointer.
- Subtracting one pointer from another.
- Comparing two pointers.

If ip points to the integer x, then *ip can occur in any context where x could, so

```
*ip = *ip + 10;
```

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1, and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.

When you subtract one pointer from another, the two pointers must have the same basic type, although you can disregard any type. Furthermore, you may compare any pointer with a null pointer constant using the equality operators (== and !=), and you may compare any object pointer with a pointer to void.

Pointer to pointer

A pointer variable is itself an object in memory, which means that a pointer can point to it. To declare a pointer to a pointer , you must use two asterisks, as in the following Example

```
char c = 'A', *cPtr = &c, **cPtrPtr = &cPtr;
```

The expression *cPtrPtr now yields the char pointer cPtr, and the value of **cPtrPtr is the char variable c. The diagram in Figure X illustrates these references.

NULL Pointers

There are times when it’s necessary to have a pointer that doesn’t point to anything. A **null pointer** is what results when you convert a **null pointer constant** to a pointer type. A null pointer constant is an integer constant expression with the value 0, or such an expression cast as the type void *.

Null pointers are implicitly converted to other pointer types as necessary for assignment operations, or for comparisons using == or !=. Hence no cast operator is necessary in the previous example.

void Pointers

A pointer to void, or **void pointer** for short, is a pointer with the type void *. As there are no objects with the type void, the type void * is used as the all-purpose pointer type. In other words, a void pointer can represent the address of any object but not its type. To access an object in memory, you must always convert a void pointer into an appropriate object pointer.

Arrays

Basic of Arrays

An array contains objects of a given type, stored consecutively in a continuous memory block.The individual objects are called the elements of an array. The elements' type can be any object type. No other types are permissible: array elements may not have a function type or an incomplete type.

An array is also an object itself, and its type is derived from its elements' type. More specifically, an array's type is determined by the type and number of elements in the array. If an array's elements have type T, then the array is called an "array of T." If the elements have type int, for example, then the array's type is "array of int." The type is an incomplete type, however, unless it also specifies the number of elements. If an array of int has 16 elements, then it has a complete object type, which is "array of 16 int elements."

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

Declarations and Usage of Arrays

The definition of an array determines its name, the type of its elements, and the number of elements in the array. The general syntax for declaring a single-dimensional array is

```
type name[ number_of_elements ];
```

The number of elements, between square brackets ([ ]), must be an integer expression whose value is greater than zero.

For example, the declaration,

```
int a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9].

```
char buffer[4*512];
```

defines an array with the name buffer, which consists of 2,048 elements of type char.

- The lower bound of an array is set at 0. C++ does not allow you to override or alter this lower bound
- Declaring a C++ array entails specifying the number of members. The number of member is equal to the upper bound plus one
- The valid range of indices extends between 0 and number_of_elements -1.

Multidimensional Arrays

A multidimensional array in C is merely an array whose elements are themselves arrays. The elements of an n-dimensional array are (n-1)-dimensional arrays. For example, each element of a two-dimensional array is a one-dimensional array. The elements of a one-dimensional array, of course, do not have an array type.

A multidimensional array declaration has a pair of brackets for each dimension:

```
char screen[10][40][80];      // A three-
dimensional array.
```

The array screen consists of the 10 elements screen[0] to screen[9]. Each of these elements is a two-dimensional array, consisting in turn of 40 one-dimensional arrays of 80 characters each. All in all, the array screen contains 32,000 elements with the type char.

Two-dimensional arrays are also called matrices. Because they are so frequently used, they merit a closer look. It is often helpful to think of the elements of a matrix as being arranged in rows and columns. Thus the matrix mat in the following definition has three rows and five columns:

```
float mat[3][5];
```

The three elements mat[0], mat[1], and mat[2] are the rows of the matrix mat. Each of these rows is an array of five float elements. Thus the matrix contains a total of 3 x 5 = 15 float elements, as the following diagram illustrates:

	0	**1**	**2**	**3**	**4**
mat[0]	0.0	0.1	0.2	0.3	0.4
mat[1]	1.0	1.1	1.2	1.3	1.4
mat[2]	2.0	2.1	2.2	2.3	2.4

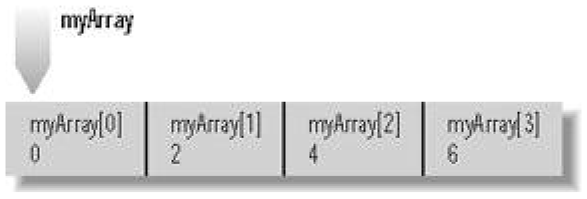
Accessing Array Elements

The subscript operator [ ] provides an easy way to address the individual elements of an array by index. If myArray is the name of an one dimensional array and i is an integer, then the expression myArray[i] designates the array element with the index i. Array elements are indexed beginning with 0. Thus, if len is the number of elements in an array, the last element of the array has the index len-1.

The following code fragment defines the array myArray and assigns a value to each element.

```
#define A_SIZE 4
long myarray[A_SIZE];
for (int i = 0;  i < A_SIZE;  ++i)
   myarray[i] = 2 * i;
```

The diagram in [link] illustrates the result of this assignment loop.



Values assigned to elements by index

To access a char element in the three-dimensional array screen, you must specify three indices. For example, the following statement writes the character Z in a char element of the array:

```
screen[9][39][79] = 'Z';
```

Initializing Arrays

If you do not explicitly initialize an array variable, the usual rules apply: if the array has automatic storage duration, then its elements have undefined values. Otherwise, all elements are initialized by default to the value 0.

- You cannot include an initialization in the definition of a variable-length array.

- If the array has static storage duration, then the array initializers must be constant expressions. If the array has automatic storage duration, then you can use variables in its initializers.
- You may omit the length of the array in its definition if you supply an initialization list. The array's length is then determined by the index of the last array element for which the list contains an initializer. For example, the definition of the array a in the previous example is equivalent to this: `int a[ ] = { 1, 2, 4, 8 }; // An array with four elements.`
- If the definition of an array contains both a length specification and an initialization list, then the length is that specified by the expression between the square brackets. Any elements for which there is no initializer in the list are initialized to zero (or NULL, for pointers). If the list contains more initializers than the array has elements, the superfluous initializers are simply ignored.
- A superfluous comma after the last initializer is also ignored.
- As a result of these rules, all of the following definitions are equivalent: `int a[4] = {1, 2}; int a[ ] = {1, 2, 0, 0}; int a[ ] = {1, 2, 0, 0, }; int a[4] = {1, 2, 0, 0, 5};`

Operations on arrays

Read the elements of a 1-dimensional array:

```
float a[10];    // declare a float array of size
10
int i;
// read the second element of the array : a[1]
scanf(“%f”,&a[1]);
// Assign an expression  to the third element of
the array
a[2] = a[1] + 5;
```

To read the value for each element of an array, you should use for statement. For example,

```
int b[10];
int i;
// Read the value for each element of the array
for(i = 0; i < 10; i++)
{
        printf(“\n Enter the value of b[%d]”, i);
        scanf(“%d”,&b[i]);
}
```

In case you do not now the exact number of elements, declare the maximum number of elements and use a variable to store the actual size of the array

```
int a[100];     // Declare the array with the
number of elements not greater than 100
        int n;                  // n is the actual
size of the array
        int i;
        printf(“\n Enter the number of elements:
“);
        scanf(“%d”,&n);
        for(i = 0; i < n; i++)
        {
                printf("\n a[%d] = ", i);
                scanf("%d",&a[i]);
        }
```

C allow you to associate initializers with specific elements . To specify a certain element to initialize, place its index in square brackets. In other words, the general form of an element designator for array elements is:

```
int a[4] = {4, 9, 22, 16};
float b[3] = {40.5, 20.1, 100};
char c[5] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’};
```

The first statement is equivalent to four assign statements

```
a[0] = 4; a[1] = 9; a[2] = 22; a[3] = 16;
```

Printing array elements

printf function are used to print the element of an array. In the following example, we print the element of array a in different ways

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int a[5];
        int i, k;
        // Read the elements of the array
        for(i = 0; i < 5; i++)
        {
                printf(“\n a[%d] = “, i);
                scanf(“%d”, &a[i]);
        }
        // print the value of element a[3]
        printf(“\n a[3] = %d”, a[3]);
        // Display all the elements of array a,
each element in a line.
        for(i = 0; i < 5; i++)
                printf(“\n%d”, a[i]);
        // Display all the elements of array a in
a line
        printf(“\n”);   // change to a new line
        for(i = 0; i < 5; i++)
                printf(“%d  “, a[i]);
        // Display all the elements of array a, k
elements in a line
        printf(“\n Enter the value of k = “);
        scanf(“%d”,&k);
        for(i = 0; i < 5; i++)
        {
                printf(“%d  “,a[i]);
                if((i+1)%k == 0)        // change
to a new line after printing k
                              //elements
                        printf(“\n”);
        }
        getch();
}
```

here is the sample session with the above program

```
 a[0] = 6
 a[1] = 14
 a[2] = 23
 a[3] = 37
 a[4] = 9
 a[3] = 37
 6
 14
 23
 37
 9
 6  14  23  37  9
 Input the value of k = 2
 6  14
 23  37
 9
```

Find the maximum value stored in the array.

The purpose of this function is to find the maximum value stored in the array

- Set up a trial minimum value. The function begins by declaring a variable named min and initializing that variable with a trial minimum value – value of the first element .
- Then the function uses a while loop to:
    - Fetch the value stored in each element in the array
    - Compare each of those values with the current value stored in the variable named max
    - Possibly replace if the value fetched from an element is algebraically greater than the current value stored in max:
    - The value fetched from the element is stored in the variable named max
    - Replacing the value that was previously stored in the variable named max by the new value from the element.
- When all of the array elements have been examined and processed in this manner, the variable named max will contain the maximum value of all the values stored in the array.

```
int a[100];
int i, n;
int max;
printf("\n Enter the size of the array: ");
```

```
scanf("%d",&n);
// Read the number of elements of the array
for(i = 0; i < n; i++)
{
        printf("\n a[%d] = ",i);
        scanf("%d",&a[i]);
}
// Find the maximum element
max = a[0];     // max is initialized by a[0]
// compare max to other elements
for(i = 1; i < n; i++)
        if(max < a[i])          //meet an element
greater than max
                max = a[i];     // replace max by
the new value from the elements.
printf("\n The maximum element of the array is:
%d", max);
```

Searching

The simplest type of searching process is the sequential search. In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the first element matching the key is found. If you are looking for an element that is near the front of the array, the sequential search will find it quickly. The more data that must be searched, the longer it will take to find the data that matches the key using this process.

here is the sample session with the above program

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int m[100], idx[100];
        int n;  // n is the actual size of the
array
        int i, k, test;
        clrscr();       // clear screen
        // Read array m
        // Read the actual size of m
        printf(" Enter the number of elements
        scanf("%d",&n);
        // Read array's elements
for(i  = 0;i<n;i++)
        {
                int temp;
        printf("\n Enter the value of m[%d] =
",i);
                scanf("%d",&temp);
                m[i] = temp;
        }
        // Read the searching key k
        printf("\n Enter the value you want to
search : ");
        scanf("%d",&k);
        // Begin searching
        test = 0;
        // Scan all the elements
        for(i = 0;i<n;i++)
                if(m[i] = = k)//Compare the
current element with the
                          //searching key k
                {
                        // save the index of the
current element
                        idx[test] = i;
test ++;
                }
        // Conclusion
        if(test > 0)
        {
                printf(“\n there are %d elements 
which has the value of %d”,test,k);
                printf(“\n Indexes of those 
elements: “);
                for(i = 0;i < test;i++)
                        printf(“%3d”,idx[i]);
        }
        else
                printf(“\n No element has the 
value %d”,k);
        getch();        // Wait until the user 
press any key
}
```

Sorting

Selection sort is a sorting algorithm, specifically an in-place comparison sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations. It works as follows:

- Find the minimum value in the list
- Swap it with the value in the first position
- Repeat the steps above for remainder of the list (starting at the second position)

Effectively, we divide the list into two parts: the sublist of items already sorted, which we build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements:

```
31 25 12 22 11
11 25 12 22 31
11 12 25 22 31
11 12 22 25 31
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int m[100];//100 is the maximum size for
array m
        int n;  // n is the number of elements
int i, j, k;
        clrscr();       // clear screen
        // Read the elements of array m
        // Read the actual size of the array

        printf(“ Enter the number of elements: “);
        scanf(“%d”,&n);
        // Read array elements
for(i  = 0;i<n;i++)
        {
                int temp;
        printf(“\n Enter the value of m[%d] =
“,i);
                scanf(“%d”,&temp);
                m[i] = temp;
        }
        // Print the array
        printf(“\n The array before sorting:
“);
        for(i=0;i<n;i++)
                printf(“%3d”,m[i]);
        // Begin to sort
        for(i = 0; i<n-1;i++)
        {
                // Put the minimum value in the
list of n-i elements
           //to the ith position
        for(j = i+1;j<n;j++)
                {
                        // compare m[i] with other
element of the sublist
// and swap m[i] and m[j] if m[j] < m[i].
                        if(m[j]<m[i])
                        {
                                int temp;
                                temp = m[j]; m[j]
= m[i]; m[i] = temp;
                        }
                }
                // Print the array after the i+1
th step of sorting process
printf("\n The array after step %d",i+1);
                for(k = 0;k < n ;k++)
printf("%3d",m[k]);
        }
        getch();        // Wait until the user
press any key.
}
```

here is the sample session with the above program

```
Enter the number of elements:  : 5
Enter the value of m[0]: 34
Enter the value of m[1]: 20
Enter the value of m[2]: 17
Enter the value of m[3]: 65
Enter the value of m[4]: 21
The array before sorting:    34 20 17 65 21
The array after step 1: 17 34 20 65 21
The array after step 2: 17 20 34 65 21
The array after step 3: 17 20 21 65 34
The array after step 4: 17 20 21 34 65
```

Pointers vs Arrays

Pointers occur in many C programs as references to arrays , and also as elements of arrays. A pointer to an array type is called an array pointer for short, and an array whose elements are pointers is called a pointer array.

Array Pointers

For the sake of example, the following description deals with an array of int. The same principles apply for any other array type, including multidimensional arrays.

To declare a pointer to an array type, you must use parentheses, as the following example illustrates:

```
int (* arrPtr)[10] = NULL; // A pointer to an
array of
                           // ten elements with
type int.
```

Without the parentheses, the declaration int * arrPtr[10]; would define arrPtr as an array of 10 pointers to int. Arrays of pointers are described in the next section.

In the example, the pointer to an array of 10 int elements is initialized with NULL. However, if we assign it the address of an appropriate array, then the expression *arrPtr yields the array, and (*arrPtr)[i] yields the array element with the index i. According to the rules for the subscript operator, the expression (*arrPtr)[i] is equivalent to *((*arrPtr)+i). Hence **arrPtr yields the first element of the array, with the index 0.

In order to demonstrate a few operations with the array pointer arrPtr, the following example uses it to address some elements of a two-dimensional array that is, some rows of a matrix:

```
int matrix[3][10];       // Array of three rows, 
each with 10 columns.
                         // The array name is a 
pointer to the first
                         // element; i.e., the 
first row.
arrPtr = matrix;         // Let arrPtr point to 
the first row of
                         // the matrix.
(*arrPtr)[0] = 5;  // Assign the value 5 to the 
first element of the
                   // first row.
                   //
arrPtr[2][9] = 6;   // Assign the value 6 to the 
last element of the
                    // last row.
                    //
++arrPtr;                // Advance the pointer to 
the next row.
(*arrPtr)[0] = 7;  // Assign the value 7 to the 
first element of the
                         // second row.
```

After the initial assignment, arrPtr points to the first row of the matrix, just as the array name matrix does. At this point you can use arrPtr in the same way as matrix to access the elements. For example, the assignment (*arrPtr)[0] = 5 is equivalent to arrPtr[0][0] = 5 or matrix[0][0] = 5.

However, unlike the array name matrix, the pointer name arrPtr does not represent a constant address, as the operation ++arrPtr shows. The increment operation increases the address stored in an array pointer by the size of one array in this case, one row of the matrix, or ten times the number of bytes in an int element.

If you want to pass a multidimensional array to a function, you must declare the corresponding function parameter as a pointer to an array type.

One more word of caution: if a is an array of ten int elements, then you cannot make the pointer from the previous example, arrPtr, point to the array a by this assignment:

```
arrPtr = a;    // Error: mismatched pointer types.
```

The reason is that an array name, such as a, is implicitly converted into a pointer to the array's first element, not a pointer to the whole array. The pointer to int is not implicitly converted into a pointer to an array of int. The assignment in the example requires an explicit type conversion, specifying the target type int (*)[10] in the cast operator:

```
arrPtr = (int (*)[10])a;     // OK
```

You can derive this notation for the array pointer type from the declaration of arrPtr by removing the identifier. However, for more readable and more flexible code, it is a good idea to define a simpler name for the type using typedef:

```
typedef int ARRAY_t[10];     // A type name for
"array of ten int elements".
ARRAY_t a,                   // An array of this
type,
        *arrPtr;             // and a pointer to
this array type.
arrPtr = (ARRAY_t *)a;       // Let arrPtr point
to a.
```

Pointer Arrays

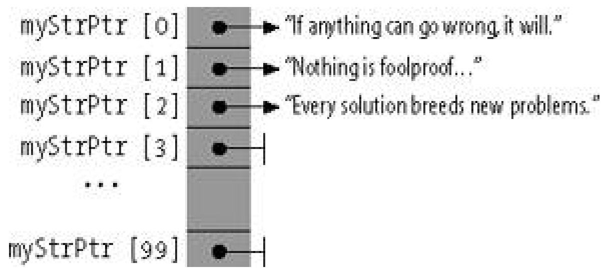
Pointer arrays that is, arrays whose elements have a pointer type are often a handy alternative to two-dimensional arrays. Usually the pointers in such an array point to dynamically allocated memory blocks.

For example, if you need to process strings, you could store them in a two-dimensional array whose row size is large enough to hold the longest string that can occur:

```
#define ARRAY_LEN 100
#define STRLEN_MAX 256
char myStrings[ARRAY_LEN][STRLEN_MAX] =
{ // Several corollaries of Murphy's Law:
  "If anything can go wrong, it will.",
  "Nothing is foolproof, because fools are so
ingenious.",
  "Every solution breeds new problems."
};
```

However, this technique wastes memory, as only a small fraction of the 25,600 bytes devoted to the array is actually used. For one thing, a short string leaves most of a row empty; for another, memory is reserved for whole rows that may never be used. A simple solution in such cases is to use an array of pointers that reference the objects in this case, the strings and to allocate memory only for the pointer array and for objects that actually exist. Unused array elements are null pointers.

```
#define ARRAY_LEN 100
char *myStrPtr[ARRAY_LEN] =    // Array of
pointers to char
{ // Several corollaries of Murphy's Law:
  "If anything can go wrong, it will.",
  "Nothing is foolproof, because fools are so
ingenious.",
  "Every solution breeds new problems."
};
```


myStrPtr [0]
"If anything can go wrong, it will."
myStrPtr [1]
"Nothing is foolproof..."
myStrPtr [2]
"Every solution breeds new problems."
myStrPtr [3]
...
myStrPtr [99]

Pointer array

The diagram in illustrates how the objects are stored in memory. The pointers not yet used can be made to point to other strings at runtime. The necessary storage can be reserved dynamically in the usual way. The memory can also be released when it is no longer needed.

Functions

Basic of C functions

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

Function declaration and definition is the area where the ANSI standard has made the most changes to C. It is now possible to declare the type of arguments when a function is declared. The syntax of function declaration also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically.

Every function is defined exactly once. A program can declare and call a function as many times as necessary.

Declaration and Usage of Function

Function Declarations

The definition of a function consists of a function head (or the declarator) and a function block . The function head specifies the name of the function, the type of its return value, and the types and names of its parameters, if any. The statements in the function block specify what the function does. The general form of a function definition is as follows:

```
//function head
type function-name(parameter declarations)
//function block
{
declarations and statements
}
```

In the function head, name is the function's name, while type (return-type) consists of at least one type specifier, which defines the type of the function's return value. The return type may be void or any object type, except array types. Furthermore, type may include the function specifier inline, and/or one of the storage class specifiers extern and static.

A function cannot return a function or an array. However, you can define a function that returns a pointer to a function or a pointer to an array.

The **parameterdeclarations** are contained in a comma-separated list of declarations of the function's parameters. If the function has no parameters, this list is either empty or contains merely the word void.

The type of a function specifies not only its return type, but also the types of all its parameters. The following listing is a simple function to calculate the volume of a cylinder.

```
// The  cylinderVolume( ) function calculates the
volume of a cylinder.
// Arguments: Radius of the base circle; height of
the cylinder.
// Return value: Volume of the cylinder.

extern double cylinderVolume( double r, double h )
{
   const double pi = 3.1415926536;     // Pi is
constant
```

```
   return  pi * r * r * h;
}
```

This function has the name cylinderVolume, and has two parameters, r and h, both with type double. It returns a value with the type double.

return statement

The return statement ends execution of the current function, and jumps back to where the function was called:

```
return [expression];
```

expression is evaluated and the result is given to the caller as the value of the function call. This return value is converted to the function's return type, if necessary.

A function can contain any number of return statements:

```
// Return the smaller of two integer arguments.
int min( int a, int b )
{
   if   ( a < b ) return a;
   else           return b;
}
```

The contents of this function block can also be expressed by the following single statement:

```
return ( a < b ? a : b );
```

The parentheses do not affect the behavior of the return statement. However, complex return expressions are often enclosed in parentheses for the sake of readability.

A return statement with no expression can only be used in a function of type void. In fact, such functions do not need to have a return statement at all. If no return statement is encountered in a function, the program flow returns to the caller when the end of the function block is reached.

Usage of Functions

The instruction to execute a function, the function call, consists of the function's name and the operator ( ). For example, the following statement calls the function maximum to compute the maximum of the matrix mat, which has r rows and c columns:

```
maximum( r, c, mat );
```

The program first allocates storage space for the parameters, then copies the argument values to the corresponding locations. Then the program jumps to the beginning of the function, and execution of the function begins with first variable definition or statement in the function block.

If the program reaches a return statement or the closing brace } of the function block, execution of the function ends, and the program jumps back to the calling function. If the program "falls off the end" of the function by reaching the closing brace, the value returned to the caller is undefined. For this reason, you must use a return statement to stop any function that does not have the type void. The value of the return expression is returned to the calling function.

Scope of Variables

One of the C language’s strengths is its flexibility in defining data storage. There are two aspects that can be controlled in C: scope and lifetime. Scope refers to the places in the code from which the variable can be accessed. Lifetime refers to the points in time at which the variable can be accessed.

Three scopes are available to the programmer:

- **extern**: This is the default for variables declared outside any function. The scope of variables with **extern** scope is all the code in the entire program.
- **static**: The scope of a variable declared static outside any function is the rest of the code in that source file. The scope of a variable declared

static inside a function is the rest of the local block.

- **auto**: This is the default for variables declared inside a function. The scope of an auto variable is the rest of the local block.

Three lifetimes are available to the programmer. They do not have predefined keywords for names as scopes do. The first is the lifetime of **extern** and **static** variables, whose lifetime is from before **main()** is called until the program exits. The second is the lifetime of function arguments and automatics, which is from the time the function is called until it returns. The third lifetime is that of dynamically allocated data. It starts when the program calls **malloc()** or **calloc()** to allocate space for the data and ends when the program calls **free()** or when it exits, whichever comes first.

Local block

A local block is any portion of a C program that is enclosed by the left brace ({) and the right brace (}). A C function contains left and right braces, and therefore anything between the two braces is contained in a local block. An if statement or a switch statement can also contain braces, so the portion of code between these two braces would be considered a local block. Additionally, you might want to create your own local block without the aid of a C function or keyword construct. This is perfectly legal. Variables can be declared within local blocks, but they must be declared only at the beginning of a local block. Variables declared in this manner are visible only within the local block. Duplicate variable names declared within a local block take precedence over variables with the same name declared outside the local block. Here is an example of a program that uses local blocks:

```
#include <stdio.h>
void main(void);
void main()
{
/* Begin local block for function main() */
int test_var = 10;
printf(“Test variable before the if statement:
```

```
%d\n”, test_var);
if (test_var > 5)
{
/* Begin local block for “if” statement */
int test_var = 5;
printf(“Test variable within the if statement:
%d\n”, test_var);
{
/* Begin independent local block (not tied to any
function or keyword) */
int test_var = 0;
printf(“Test variable within the independent local
block:%d\n”, test_var);
}
/* End independent local block */
}
/* End local block for “if” statement */
printf(“Test variable after the if statement:
%d\n”, test_var);
}
/* End local block for function main() */
```

This example program produces the following output:

```
Test variable before the if statement: 10
Test variable within the if statement: 5
Test variable within the independent local block:


0
Test variable after the if statement: 10
```

Notice that as each test_var was defined, it took precedence over the previously defined test_var. Also notice that when the if statement local block had ended, the program had reentered the scope of the original test_var, and its value was 10.

Functions and Storage Class Specifiers

The function in the listing above is declared with the storage class specifier extern. This is not strictly necessary, since extern is the default storage class for functions. An ordinary function definition that does not contain a static or inline specifier can be placed in any source file of a program. Such a function is available in all of the program's source files, because its name is an external identifier. You merely have to declare the function before its first use in a given translation unit. Furthermore, you can arrange functions in any order you wish within a source file. The only restriction is that you cannot define one function within another. C does not allow you to define "local functions" in this way.

You can hide a function from other source files. If you declare a function as static, its name identifies it only within the source file containing the function definition. Because the name of a static function is not an external identifier, you cannot use it in other source files. If you try to call such a function by its name in another source file, the linker will issue an error message, or the function call might refer to a different function with the same name elsewhere in the program.

The function **printArray( )** in the following listing might well be defined using static because it is a special-purpose helper function, providing formatted output of an array of float variables.

```
// The static function printArray( ) prints the
elements of an array
// of float to standard output, using printf( ) to
format them.
// Arguments:    An array of float, and its
length.
// Return value: None.

static void printArray( const float array[ ], int
n )
{
  for ( int i=0; i < n; ++i )
  {
    printf( "%12.2f", array[i] );     // Field
width: 12; decimal places: 2
    if ( i % 5 == 4 ) putchar( '\n' );// New line
after every 5 numbers
  }
  if ( n % 5 != 0 ) putchar( '\n' ); // New line
at the end of the output
}
```

If your program contains a call to the **printArray()** function before its definition, you must first declare it using the static keyword:

```
static void printArray(const float [ ], int);

int main( )
{
  float farray[123];
  /* ... */
  printArray( farray, 123 );
  /* ... */
}
```

Function prototype

A function prototype in C++ is a declaration of a function that omits the function body but does specify the function's name, arity, argument types and return type. While a function definition specifies what a function does, a function prototype can be thought of as specifying its interface. Just like a blueprint, the prototype tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed. The general format for a prototype is simple:

```
type function_name ( arg_type arg1, ..., arg_type
argN );
```

arg_type just means the type for each argument -- for instance, an int, a float, or a char. It's exactly the same thing as what you would put if you were declaring a variable.

There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that do not return values have a return type of void. Lets look at a function prototype:

```
int mult ( int x, int y );
```

This prototype specifies that the function mult will accept two arguments, both integers, and that it will return an integer. Do not forget the trailing semi-colon. Without it, the compiler will probably think that you are trying to write the actual definition of the function.

When the programmer actually defines the function, it will begin with the prototype, minus the semi-colon. Then there should always be a block with the code that the function is to execute, just as you would write it for the main function. Any of the arguments passed to the function can be used as if they were declared in the block.

Lets look at an example program:

```
#include <stdio.h>
#include <conio.h>
int mult ( int x, int y );
int main()
{
  int x;
  int y;
  printf("Please input two numbers to be
multiplied: ");
  scanf("%d%d", &x,&y);
  printf("The product of your two numbers is
%d\n", mult ( x, y )) ;
  return 0;
  getch();
}
int mult ( int x, int y )
{
  return x * y;
}
```

Parameters passing

The parameters of a function are ordinary local variables. The program creates them, and initializes them with the values of the corresponding arguments, when a function call occurs. Their scope is the function block. A function can change the value of a parameter without affecting the value of the argument in the context of the function call. In the following listing, the **factorial( )** function, which computes the factorial of a whole number, modifies its parameter n in the process.

```
//factorial( ) calculates n!, the factorial of a
non-negative number n.
// For n > 0, n! is the product of all integers
from 1 to n inclusive.
// 0! equals 1.
// Argument:     A whole number, with type
unsigned int.
// Return value: The factorial of the argument,
with type long double.

long double factorial(register unsigned int n)
{
  long double f = 1;
  while ( n > 1 )
    f *= n--;
  return f;
}
```

Although the factorial of an integer is always an integer, the function uses the type long double in order to accommodate very large results. As the above listing illustrates, you can use the storage class specifier register in declaring function parameters. The register specifier is a request to the compiler to make a variable as quickly accessible as possible. No other storage class specifiers are permitted on function parameters.

Arrays as Function Parameters

If you need to pass an array as an argument to a function, you would generally declare the corresponding parameter in the following form:

```
type name[ ]
```

Because array names are automatically converted to pointers when you use them as function arguments, this statement is equivalent to the declaration:

```
type *name
```

When you use the array notation in declaring function parameters, any constant expression between the brackets ([ ]) is ignored. In the function block, the parameter name is a pointer variable, and can be modified. Thus the function **addArray()** in the following listing modifies its first two parameters as it adds pairs of elements in two arrays.

```
// addArray( ) adds each element of the second
array to the
// corresponding element of the first (i.e.,
"array1 += array2", so to speak).
// Arguments:    Two arrays of float and their
common length.
// Return value: None.

void addArray( register float a1[ ], register
const float a2[ ], int len )
{
  register float *end = a1 + len;
  for ( ; a1 < end; ++a1, ++a2 )
```

```
    *a1 += *a2;
}
```

An equivalent definition of the **addArray()** function, using a different notation for the array parameters, would be:

```
void addArray( register float *a1, register const
float *a2, int len )
{  /* Function body as earlier. */  }
```

An advantage of declaring the parameters with brackets ([ ]) is that human readers immediately recognize that the function treats the arguments as pointers to an array, and not just to an individual float variable. But the array-style notation also has two peculiarities in parameter declarations :

- In a parameter declaration and only there C allows you to place any of the type qualifiers const, volatile, and restrict inside the square brackets. This ability allows you to declare the parameter as a qualified pointer type.
- Furthermore, in C you can also place the storage class specifier static, together with a integer constant expression, inside the square brackets. This approach indicates that the number of elements in the array at the time of the function call must be at least equal to the value of the constant expression.

Here is an example that combines both of these possibilities:

```
int func( long array[const static 5] )
{ /* ... */ }
```

In the function defined here, the parameter array is a constant pointer to long, and so cannot be modified. It points to the first of at least five array elements.

In the following listing, the **maximum( )** function's third parameter is a two-dimensional array of variable dimensions.

```
// The function maximum( ) obtains the greatest
value in a
// two-dimensional matrix of double values.
// Arguments:    The number of rows, the number of
columns, and the matrix.
// Return value: The value of the greatest
element.

double maximum( int nrows, int ncols, double
matrix[nrows][ncols] )
{
  double max = matrix[0][0];
  for ( int r = 0; r < nrows; ++r )
    for ( int c = 0; c < ncols; ++c )
      if ( max < matrix[r][c] )
        max = matrix[r][c];
  return max;
}
```

The parameter matrix is a pointer to an array with ncols elements.

Pointers as Function Parameters

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called swap. It is not enough to write

```
swap(a, b);
```

where the swap function is defined as

```
void swap(int x, int y) /* WRONG */
{
int temp;
temp = x;
x = y;
y = temp;
}
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps copies of a and b.

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

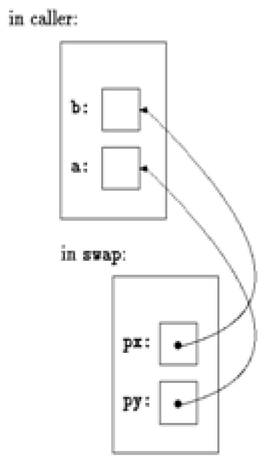
```
 swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py) /* interchange *px and
*py */
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
```

{

Pictorially in [link]



swap function with pointer parameters

Strings

Basic of strings

A **string** is a continuous sequence of characters terminated by '\0', the null character. The **length** of a string is considered to be the number of characters excluding the terminating null character. There is no string type in C, and consequently there are no operators that accept strings as operands.

Instead, strings are stored in arrays whose elements have the type **char** or wchar_t. Strings of wide characters that is, characters of the type wchar_tare also called **wide strings**. The C standard library provides numerous functions to perform basic operations on strings, such as comparing, copying, and concatenating them.

Declarations and Uses of Strings

You can initialize arrays of char or wchar_t using string literals. For example, the following two array definitions are equivalent:

```
char str1[30] = "Let's go";     // String length:
8; array length: 30.

char str1[30] = { 'L', 'e', 't', '\'', 's',' ',
'g', 'o', '\0' };
```

An array holding a string must always be at least one element longer than the string length to accommodate the terminating null character. Thus the array str1 can store strings up to a maximum length of 29. It would be a mistake to define the array with length 8 rather than 30, because then it wouldn't contain the terminating null character.

If you define a character array without an explicit length and initialize it with a string literal, the array created is one element longer than the string length. An Example

```
char str2[ ] = " to London!";// String length: 11
(note leading space);
                              // array length: 12.
```

The following statement uses the standard function **strcat()** to append the string in str2 to the string in str1. The array str1 must be large enough to hold all the characters in the concatenated string.

```
#include <string.h>

char str1[30] = "Let's go";
char str2[ ] = " to London!";

/* ... */

strcat( str1, str2 );
puts( str1 );
```

The output printed by the **puts()** call is the new content of the array str1:

Let's go to London!

The names str1 and str2 are pointers to the first character of the string stored in each array. Such a pointer is called a **pointer to a string**, or a **string pointer** for short. String manipulation functions such as **strcat()** and **puts()** receive the beginning addresses of strings as their arguments. Such functions generally process a string character by character until they reach the terminator, '\0'. The function in is one possible implementation of the standard function **strcat()**. It uses pointers to step through the strings referenced by its arguments.

Built-in Functions for Character and String Processing

Character Processing Functions

The standard library provides a number of functions to classify characters and to perform conversions on them. The header ctype.h declares such functions for byte characters, with character codes from 0 to 255.

The results of these functions, except for **isdigit()** and **isxdigit()**, depends on the current locale setting for the locale category LC_CTYPE. You can query or change the locale using the **setlocale()** function.

Character Classification Functions

The functions listed in [link] test whether a character belongs to a certain category. Their return value is nonzero, or true, if the argument is a character code in the given category.

Category	**Functions**
Letters	isalpha( )
Lowercase letters	islower( )
Uppercase letters	isupper( )
Decimal digits	isdigit( )
Hexadecimal digits	isxdigit( )
Letters and decimal digits	isalnum( )
Printable characters (including whitespace)	isprint( )
Printable, non-whitespace characters	isgraph( )
Whitespace characters	isspace( )

Whitespace characters that separate words in a line of text	isblank( )
Punctuation marks	ispunct( )
Control characters	iscntrl( )

Character classification functions

The functions **isgraph()** and **iswgraph()** behave differently if the execution character set contains other byte-coded, printable, whitespace characters (that is, whitespace characters which are not control characters) in addition to the space character (' '). In that case, **iswgraph()** returns false for all such printable whitespace characters, while **isgraph()** returns false only for the space character (' ').

Case Mapping Functions

The functions listed in [link] yield the uppercase letter that corresponds to a given lowercase letter, and vice versa. All other argument values are returned unchanged.

Conversion	**Functions in ctype.h**
Upper- to lowercase	tolower( )
Lower- to uppercase	toupper( )

Character conversion functions

String Processing Functions

A **string** is a continuous sequence of characters terminated by '\0', the string terminator character. The length of a string is considered to be the number of characters before the string terminator. Strings are stored in arrays whose elements have the type char or wchar_t. Strings of wide characters that is, characters of the type wchar_tare also called **wide strings**.

C does not have a basic type for strings, and hence has no operators to concatenate, compare, or assign values to strings. Instead, the standard library provides numerous functions, listed in

[link] to perform these and other operations with strings. The header **string.h** declares the functions for conventional strings of char. The names of these functions begin with str.

Like any other array, a string that occurs in an expression is implicitly converted into a pointer to its first element. Thus when you pass a string as an argument to a function, the function receives only a pointer to the first character, and can determine the length of the string only by the position of the string terminator character.

Purpose	**Functions**
Find the length of a string.	strlen( )
Copy a string.	strcpy( ), strncpy( )
Concatenate strings.	strcat( ), strncat( )

Purpose	**Functions**
Compare strings.	strcmp( ), strncmp( ), strcoll( )
Transform a string so that a comparison of two transformed strings using strcmp( ) yields the same result as a comparison of the original strings using the locale-sensitive function strcoll( ).	strxfrm( )
In a string, find:	
- The first or last occurrence of a given character	strchr( ), strrchr( )
- The first occurrence of another string	strstr( )
- The first occurrence of any of a given set of characters	strcspn( ), strpbrk( )
- The first character that is not a member of a given set	strspn( )
Parse a string into tokens	strtok( )

String-processing functions

Example:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>     // You must declare the
library string.h
// to use functions strcpy, strcmp...
```

```
void main()
{
        char str1[10] = “abc”;
        char str2[10] = “def”;
        clrscr();
        printf(“ str1: %s”,str1);
        printf(“\n str2: %s”,str2);
        printf(“\n strcmp(str1,str2) =
%d”,strcmp(str1,str2));
        printf(“\n strcat(str1,str2) =
%s”,strcat(str1,str2));
        printf(“\n str1: %s”,str1);
        printf(“\n str2: %s”,str2);
        printf(“\n strcpy(str1,str2) =
%s”,strcpy(str1,str2));
        printf(“\n str1: %s”,str1);
        printf(“\n str2: %s”,str2);
        strcpy(str1,”ab”);
        strcpy(str2,”abc”);
        printf(“\n strcmp(str1,str2) =
%d”,strcmp(str1,str2));
        getch();
}
```

here is the sample session with the above program

```
str1: abc
str2: def
strcmp(str1,str2) = -3
strcat(str1,str2) = abcdef
str1: abcdef
str2: def
strcpy(str1,str2) = def
str1: def
```

```
str2: def
strcmp(str1,str2) = -3
```

Structures

Introduction

A structure type can contain a number of dissimilar data objects within it. Unlike a simple variable (which contains only one data object) or an array (which, although it contains more than one data item, only contains items of a single data type),a structure is a collection of related data of different types. a name, for example, might be array of characters, an age might be integer. A structure representing a person, say, could contain both a name and an age, each represented in the appropriate format.

Declarations and Usage of Structures

Until now, all the data that we have dealt with has been either of a basic type such as char, int and double…, or an array of those types. However, there are many situations in real life where a data item needs to be made up from other more basic types. We could do this with an array if the constituent types were all the same, but often they are different. For example, suppose we want to record the details of each student in a class. The detail of each student might be as follow:

- A unique student number, which could be represented as a string (an array of **char**).
- The student’s name, which could be represented as a string (an array of **char**).
- Final mark for the Introduction to computer science course, which is a floating point value (a **float**).

Creating Structures as New Data Types

The definition of a structure type begins with the keyword struct, and contains a list of declarations of the structure's members, in braces:

```
struct structTag
   {
          <list of members>;
   };
```

Note: Definition ends with semicolon (;)

Example:
The three components above can be placed in a structure declared like this:

```
struct Student
{
    char StudentID[10];
    char name[30];
    float markCS ;
};
```

The keyword **struct** introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a structure tag may follow the word **struct** (as with Student here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces. The variables named in a structure are called members. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

Creating variable of a struct type

Structure types are not considered a variable declaration, just definition of a new type, so they cannot store anything until we declare variable of this type. Here is how we would create:

```
type_name_of_struct   name_of_variable;
```

Example:
Creating three variables a, b, c of the Student type:

```
Student a, b, c;
```

Creating an array of the Student type:

```
Student studentCS[50];
```

A member of a structure may have any desired complete type, including previously defined structure types. They must not be variable-length arrays, or pointers to such arrays. For instance, now we want to record more information of students, for example their date of birth, which comprises the day, month and year. So first, let's start with the date, because that is a new type that we may be able to use in a variety of situations. We can declare a new type for a **Date** thus:

```
struct Date
    {
      int day;
      int month;
      int year;
    };
```

We can now use this **Date** type, together with other types, as members of a **Student** type which we can declare as follows:

```
struct Student
    {
       char studentID[10];
       char name[30];
       float markCS ;
       Date  dateOfBirth;
    };
```

Or

```
struct Student
{
    char studentID[10];
    char name[30];
    float markCS;
    struct Date {
                  int day;
                  int month;
                  int year;
                } dateOfBirth;
};
```

We can also declare structured variables when we define the structure itself:

```
struct  Student
{
    char studentID[10];
    char name[30];
    float markCS ;
    Date  dateOfBirth;
} a, b, c;
```

C permits to declare untagged structures that enable us to declare structure variables without defining a name for their structures. For example, the following structure definition declares the variables a, b, c but omits the name of the structure:

```
struct
{
    char studentID[10];
    char name[30];
    float markCS ;
    Date  dateOfBirth;
} a, b, c;
```

A structure type cannot contain itself as a member, as its definition is not complete until the closing brace (}). However, structure types can and often do contain pointers to their own type. Such self-referential structures are used in implementing linked lists and binary trees, for example. The following example defines a type for the members of a singly linked list:

```
struct List
{   struct Student stu;     // This record's data.
    struct List *pNext;    // A pointer to the
next student.
};
```

Referencing Structure Members with the Dot Operator

Whenever we need to refer to the members of a structure, we normally use the dot operator.

For example, if we wanted to access the number member of newStudent we could do so as follows:

newStudent.studentID

We can then access the member as if it were a normal variable. For instance, we can write to this member as follows.

newStudent.studentID= “C0681008”;

We can also read from the member in a similar fashion.

```
printf("Student identification: %s",
newStudent.studentID);
```

The following code outputs the contents of an **Student** structure.

```
printf("Student Details\n");
printf("Identification: %s\n",
newStudent.studentID);
printf("Name: %s\n", newStudent.name);
printf("Mark: %.2f\n", newStudent.markCS);
printf("Date of Birth: %i/%i/%i\n",
             newStudent.dateOfBirth.day,
             newStudent.dateOfBirth.month,
             newStudent.dateOfBirth.year
);
```

Suppose we wish to input the details of this employee from a user. We could do so as follows.

```
Student newStudent;
printf("Enter student identification: ");
scanf("%s", &newStudent.studentID);
printf("Enter student name: ");
fflush(stdin);gets(newStudent.name);
printf("Enter mark for Introduction to computer
science course: ”);
scanf("%f", &newStudent.markCS);
printf("Enter birth date (dd/mm/yyyy): ");
```

```
scanf("%i/%i/%i",
&newStudent.dateOfBirth.day,
&newStudent.dateOfBirth.month,
&newStudent.dateOfBirth.year
);
```

Initializing Structure Variables

When we declare a new variable of a basic data type we can initialize its value at declaration. We can also initialize structure variables at declaration as shown below.

```
Student newStudent = {
            "C0681008",
            "Cao Anh Huy",
            8.50,
            {1, 2, 1985}
        };
```

Notice how we include the initialization values in curly brackets, just like when we initialize an array. Furthermore, we include the values for any nested structure type (in this case the **dateOfBirth** member is a nested structure), in a further set of curly brackets.

Copying Structure Variables

One of the most convenient features of structures is that we can copy them in a single assignment operation. This is unlike an array, which must be copied item-by-item. The name of a structure variable when it appears on its own represents the entire structure. If a structure contains an array as a member, that array is copied if the entire structure is copied.

```
Student newStudent1, newStudent2;
// Get the values for newStudent2
...
```

```
// Copy newStudent2's value to newStudent1
newStudent1 = newStudent2;
```

Comparing Values of Structures

We cannot compare structures in a single operation. If we wish to compare the values of two structure variables, we need to compare each of their members.

Arrays of Structures

Just as we can have an array of basic data types, we can also have an array of structures. Suppose that we created an array of **Student** structures as follows. We could then copy **newStudent** into each position in the array.

```
Student  students[100];
Student  newStudent;
int i;
for (i=0; i<100; i++)
{
// Get the values for newStudent
...
// Copy into the next position in the array
students[i] = newStudent;
}
```

Operations on Structures

Passing Structures to and from Functions

Structures can be passed to functions just like any other data type. Functions can also return structures, just as they can return any basic type. Structures can be also be passed to functions by reference.

Just like passing variable of a basic data type, when we pass a structure as an argument to a function, a copy is made of the entire structure. Structures are passed by value. We can easily take the code that output a student and put it into a function as follows.

```
void outputStudent(Student stu)
{
printf("Student Details\n");
printf("Identification: %s\n", stu.studentID);
printf("Name: %s\n", stu.name);
printf("Mark: %.2f\n, stu.markCS);
printf("Date of Birth: %i/%i/%i\n",
           stu.dateOfBirth.day,
           stu.dateOfBirth.month,
           stu.dateOfBirth.year
      );
}
```

If we had an array of 100 students and wanted to output them this would be straightforward:

```
Student students[100];
int i;
...
for (i=0; i<100; i++) {
   outputStudent(students[i]);
}
```

We could similarly place the code to input a student into a function, but now we have a problem. The function can return a structure of type **Student** as follows.

```
Student inputStudent()
{
   Student tempStudent;
   printf("Enter Student identification: ");
   scanf("%s", &tempStudent.studentID);
   printf("Enter Student name: ");
   fflush(stdin);gets(tempStudent.name);
   printf("Enter final mark: ");
   scanf("%f", &tempStudent.markCS);
   printf("Enter birth date (dd/mm/yyyy):");
   scanf("%i/%i/%i",
         &tempStudent.dateOfBirth.day,
         &tempStudent.dateOfBirth.month,
         &tempStudent.dateOfBirth.year
        );
   return tempStudent;
}
```

In the example above we are filling the structure variable **tempStudent** with values. At the end of the function, the value of **tempStudent** is returned as the return value of the function. The code to input 100 students can now be modified to use this function:

```
Student students[100];
int i;
for (i=0; i<100; i++) {
    students[i] = inputStudent();
}
```

The Arrow Operator

In order to dereference a pointer we would normally use the dereferencing operator (*) and if we our pointer was to a structure, we could subsequently use the dot '.' operator to refer to a member of the structure. Suppose we have declared a pointer which could be used to point to a structure of type employee as follows.

```
Student stuVariable;
Student *stuPtr;
stuPtr = &stuVariable;
```

To refer to the student identification we could say:

```
(*stuPtr).studentID
```

Note that the brackets are necessary because the dereference operator has lower precedence than the dot operator. This form of syntax is a little cumbersome, so another operator is provided to us as a convenient shorthand:

```
stuPtr->studentID
```

This method of accessing the number member through a pointer is completely equivalent to the previous form. The '->' operator is called the indirect member selection operator or just the arrow operator and it is almost always used in preference to the previous form.

Passing Structures by Reference

Passing structures to a function using pass-by-value can be simple and successful for simple structures so long as we do not wish to do so repeatedly. But when structures can contain a large amount of data (therefore occupying a large chunk of memory) then creating a new copy to pass to a function can create a burden on the memory of the computer. If we were going to do this repeatedly (say several thousand times within the running of a computer) then there would also be a cost in time to copy the structure for each function call.

In the example at the beginning of this section we created and filled a structure variable called tempStudent. When the function ended it returned the value of tempStudent. The same inefficiency exists with the return value from the function, where the Student structure must be copied to a local variable at the function call.

Whether such inefficiencies are of any significance or not depends on the circumstances and on the size of the structure. Each Student structure probably occupies about 50 bytes, so this is a reasonably significant amount of memory to be copying each time the output function is called or each time the input function returns, especially if this is happening frequently.

A better solution would be to pass the Student structure by reference, which means we will pass a pointer to the structure.

We can now revise the input function by passing an Student structure by reference using a pointer. Because the function is no longer returning an Student structure, we can also enhance the function to return a Boolean status indicating whether an Student structure was successfully read or not. We can enhance our function to do some better error checking. Below is the revised version.

```
bool inputStudent(Student *stuPtr)
{
printf("Enter Student identification: ");
if (scanf("%s", &stuPtr->studentID) != 1) return
false;
printf("Enter Student name: ");
fflush(stdin);gets(stuPtr->name);
printf("Enter mark: ");
if (scanf("%f", &stuPtr->markCS) != 1) return
false;
printf("Enter birth date: ");
if (scanf("%i/%i/%i",&stuPtr->dateOfBirth.day,
&stuPtr->dateOfBirth.month,&stuPtr-
>dateOfBirth.year) != 3)
   return false;
return true;
}
```

The code to input 100 students can now be revised as follows.

```
Student students[100];
int i;
for (i=0; i<100; i++)
{
   while (!inputStudent(&students[i]))
   {
printf("Invalid student details - try again!\n");
fflush(stdin);
```

```
   }
}
```

As a final example, consider a function to give s student a mark rise. The function takes two parameters. The first is an Student structure passed by reference, (a pointer to an Student structure) and the second is the increase of mark.

```
void markRise(Student *stuPtr, float increase)
{
stuPtr->markCS += increase;
}
```

What use is such a function? Having input many students into an array, we might then wish to give certain students a mark rise. For each student we can easily call this function, passing a pointer to the appropriate Student structure.

Enumerated Types

Another way of creating a new type is by creating an enumerated type. With an enumerated type we build a new type from scratch by stating which values are in the type. The syntax for an enumerated type is as follows.

```
enum TypeIdentifier { list... };
```

Here is an example of a definition of an enumerated type that can be used to refer to the days of the week.

```
enum DayOfWeek {sun, mon, tue, wed, thu, fri,
sat};
```

Just like when we define a structure type, defining an enumerated type does not give us any space to store information. We use the type like a template to create variables of that type.

For instance we can create a variable of type DayOfWeek as follows.

```
DayOfWeek nameOfDay;
```

With variables of enumerated types we can do almost anything we could do with a variable of a basic data type. For instance we can assign a value as follows.

```
nameOfDay = tue;
```

Note that **tue** is a literal value of type **DayOfWeek** and we do not need to place quotes around it.

The values in **DayOfWeek** are ordered and each has an equivalent **int** value; sun==0, mon==1, and so on. The value of sun is less than the value of **wed** because of the order they were presented in the list of values when defining the type. We can compare two values of enumerated types as follows:

```
DayOfWeek day1, day2;
// Get day values
...
if(day1 < day2) {
...
}
```

Here is another example that uses enumerated types.

```
#include <stdio.h>
#include <conio.h>
enum TrafficLight {red, orange, green};
int main()
 {
   TrafficLight light;
   printf("Please enter a Light Value: (0)Red
(1)Orange (2)Green:\n");
   scanf("%i", &light);
   switch(light)
    {
      case red:
```

```
      printf("Stop!\n");
      break;
   case orange:
      printf("Slow Down\n");
      break;
   case green:
      printf("Go\n");
 }
getch();
}
```

Files

Basics and Classification of Files

When reading input from the keyboard and writing output to the monitor you have been using a special case of file I/O (input/output). You already know how to read and write text data, as you have been doing it every time you use **scanf()** and **printf()**. All you need to do now is learn how to direct I/O to file other than from your keyboard or to your monitor.

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files:

- **Text files** are any files that contain only ASCII characters. Examples include C source code files, HTML files, and any file that can be viewed using a simple text editor.
- **Binary files** are any files that created by writing on it from a C-program, not by an editor (as with text files). Binary files are very similar to arrays of records, except the records are in a disk file rather than in an array in memory. Because the records in a binary file are on disk, you can create very large collections of them (limited only by your available disk space). They are also permanent and always available. The only disadvantage is the slowness that comes from disk access time.

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time). A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signaled the intention to process a text file.

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

- No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
- C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files – they a generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

For all file operations you should always follow the 5-step plan as outlined below.

1. Declare file pointer.
2. Attach the file pointer to the file (open file).
3. Check file opened correctly.
4. Read or Write the data from or to the file.
5. Close the file.

Operations on Files

Declarations

In C, we usually create variables of type **FILE** * to point to a file located on the computer.

```
FILE *file_pointer_name;
```

Example

```
FILE * f1, * f2;
```

Open Files

First things first: we have to open a file to be able to do anything else with it. For this, we use **fopen** function, like all the I/O functions, is made available by the **stdio.h** library. The **fopen()** function prototype is as follows.

```
FILE *fopen(char *filename, char *mode);
```

In the above prototype, there are two arguments:

- **filename** is a string containing the name of the file to be opened. So if your file sits in the same directory as your C source file, you can simply enter the filename in here - this is probably the one you'll use most.
- **mode** determines how the file may be accessed.

Mode	**Meaning**
“r”	Open a file for read only, starts at beginning of file (default mode).
“w”	Write-only, truncates existing file to zero length or create a new file for writing.
“a”	Write-only, starts at end of file if file exists,otherwise creates a new file for writing.
“r+”	Open a file for read-write, starts at beginning of file. If the file is not exist, it will cause an error.
“w+”	Read-write, truncates existing file to zero length or creates a new file for reading and writing.
“a+”	Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.

So there are 12 different values that could be used: **"rt", "wt", "at", "r+t", "w+t", "a+t" and "rb", "wb", "ab", "r+b", "w+b", "a+b".**

Character	**Type**
“t”	Text File
“b”	Binary File

Note: When work with the text file, you also can use only "r", "w", "a", "r", "w”, "a", instead of "rt", "wt", "at", "r+t", "w+t", "a+t" respectively.

Example

```
FILE *f1, *f2, *f3, *f4;
```

To open text file c:\abc.txt for ready only:

```
f1 = fopen("c:\\abc.txt", "r");
```

To open text file c:\list.dat for write only:

```
f2 = fopen("c:\\list.dat", "w");
```

To open text file c:\abc.txt for read-write:

```
f3 = fopen("c:\\abc.txt", "r+");
```

To open binary file c:\liststudent.dat for write only:

```
f4 = fopen("c:\\liststudent.dat", "wb");
```

The file pointer will be used with all other functions that operate on the file and it must never be altered or the object it points to.

File checking

```
if (file_pointer_name == NULL)
{
printf("Error opening file.");
<Action for error >
}
else
{
<Action for success>
}
```

Before using an input/output file it is worth checking that the file has been correctly opened first. A call to fopen() may result in an error due to a number of reasons including:

- A file opened for reading does not exist;
- A file opened for reading is read protected;
- A file is being opened for writing in a folder or directory where you do not have write access.

If the operation is successful, **fopen**() returns an address which can be used as a stream. If a file is not successfully opened, the value **NULL** is returned. An error opening a file can occur if the file was to be opened for reading and did not exist, or a file opened for writing could not be created due to lack of disk space. It is important to always check that the file has opened correctly before proceeding in the program.

Example:

```
FILE *fp;
if ((fp = fopen("myfile", "r")) ==NULL){
  printf("Error opening file\n");
  exit(1);
}
```

Once a file has been opened, depending upon its mode, you may read and/or write bytes to or from it.

Access to Text Files

Write data to text files

When writing data to text files, C provides three functions: **fprintf()**, **fputs()**, **fputc()**.

The **fprintf()** function prototype is as follows:

```
int fprintf(FILE *fp, char *format, ...);
```

This function writes to the file specified by file pointer fp a sequence of data formatted as the format argument specifies. After the format parameter, the function expects at least as many additional arguments as specified in format. Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.

Return value: On success, the total number of characters written is returned. On failure, a negative number is returned.

Example:

```
#include <stdio.h>

int main ()
{
   FILE * fp;
   int n;
   char name [50];

   fp = fopen ("myfile.txt","w");
   for (n=0 ; n<3 ; n++)
   {
     puts ("Please, enter a name: ");
     gets (name);
     fprintf (fp, "Name %d [%-10.10s]\n",n,name);
   }
   fclose (fp);
   return 0;
}
```

This example prompts 3 times the user for a name and then writes them to myfile.txt each one in a line with a fixed length (a total of 19 characters + newline). Two format tags are used: %d : signed decimal integer, %-10.10s : left aligned (-), minimum of ten characters (10), maximum of ten characters (.10), String (s).

Assuming that we have entered John, Jean-Francois and Yoko as the 3 names, myfile.txt would contain:

myfile.txt
Name 1 [John]
Name 2 [Jean-Franc]
Name 3 [Yoko]

The **fputc()** function prototype is as follows.

```
int fputc(int character, FILE *fp);
```

The **fputc()** function writes a character to the file associated with fp. The character is written at the current position of the fp as indicated by the internal position indicator, which is then advanced one character

Return value: If there are no errors, the same character that has been written is returned.If an error occurs, EOF is returned and the error indicator is set.

Example:
Write the program that creates a file called alphabet.txt and writes ABCDEFGHIJKLMNOPQRSTUVWXYZ to it.

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  char c;

  fp = fopen ("alphabet.txt","w");
  if (fp!=NULL)
  {
    for (c = 'A' ; c <= 'Z' ; c++)
    {
      fputc ((int) c , fp);
    }
    fclose (fp);
  }
  return 0;
}
```

The **fputs()** function prototype is as follows.

```
int fputs(char *str,FILE *fp);
```

The **fputs()** function writes the string pointed to by str to the file associated with fp.

Return value: On success, a non-negative value is returned. On error, the function returns EOF. The null that terminates str is not written and it does not automatically append a carriage return/linefeed sequence.

Example:
Write the program allows to append a line to a file called myfile.txt each time it is run.

```
#include <stdio.h>
int main ()
{
   FILE * fp;
   char name [50];

   puts ("Please, enter a name: ");
   gets (name);
   fp = fopen ("myfile.txt","a");
   fputs (name,fp);
   fclose (fp);
   return 0;
}
```

Read data from text files

When reading data from text files, C provides three functions: **fscanf()**,**fgetc()**, **fgets()**.

The **fscanf()** function prototype is as follows.

```
int fscanf(FILE *fp, char *format, ...);
```

This function reads data from the file specified by file pointer fp and stores them according to the parameter format into the locations pointed by the additional arguments. The additional arguments should point to already allocated objects of the type specified by their corresponding format tag within the format string.

Return value: On success, the function returns the number of items successfully read. This count can match the expected number of readings or be less -even zero- in the case of a matching failure. In the case of an input failure before any data could be successfully read, EOF is returned.

Example:
Read an integer number and a character from file associated with a file pointer fp and stores them to two variables a and c.

```
fscanf(fp, "%d %c",&a, &c);
```

Example:

```
#include <stdio.h>
int main ()
{
  char str [80];
  float f;
  FILE * fp;

  fp = fopen ("myfile.txt","w+");
  fprintf (fp, "%f %s", 3.1416, "PI");
  rewind (fp);
  fscanf (fp, "%f", &f);
  fscanf (fp, "%s", str);
  fclose (fp);
  printf ("I have read: %f and %s \n",f,str);
  return 0;
}
```

This sample code creates a file called myfile.txt and writes a float number and a string to it. Then, the stream is rewinded and both values are read with fscanf. It finally produces an output similar to:

```
I have read: 3.141600 and PI
```

feof() function

```
int feof(FILE *fp);
```

This function check if End-of-File indicator associated with fp is set

Return value: A non-zero value is returned in the case that the End-of-File indicator associated with the fp is set. Otherwise, a zero value is returned.

Example:
Create a text file called fscanf.txt in Notepad with this content:

```
0       1       2       3       4
5       6       7       8       9
10      11      12      13
```

Remember how **scanf** stops reading input when it encounters a space, line break or tab character? fscanf is just the same. So if all goes to plan, this example should open the file, read all the numbers and print them out:

```
#include <stdio.h>
int main() {
  FILE *fp;
  int numbers[30];
  /* make sure it is large enough to hold all the
data! */
  int i,j;

  fp = fopen("fscanf.txt", "r");

  if(fp==NULL) {
    printf("Error: can't open file.\n");
    return 1;
  }
  else {
    printf("File opened successfully.\n");

    i = 0 ;

    while(!feof(fp)) {
      /* loop through and store the numbers into
the array */
      fscanf(fp, "%d", &numbers[i]);
      i++;
    }

    printf("Number of numbers read: %d\n\n", i);
    printf("The numbers are:\n");

    for(j=0 ; j<i ; j++) { /* now print them out
one by one */
      printf("%d\n", numbers[j]);
    }

    fclose(fp);
    return 0;
  }
}
```

fflush() function

Same as **scanf()**, before using **fscanf()** to read the character or string from the file, we need use **fflush()**.The **fflush()** function prototype is as follows.

```
int fflush(FILE *fp)
```

If the given file that specified by fp was open for writing and the last I/O operation was an output operation, any unwritten data in the output buffer is written to the file. If the file was open for reading, the behavior depends on the specific implementation. In some implementations this causes the input buffer to be cleared. If the argument is a null pointer, all open files are flushed. The files remains open after this call. When a file is closed, either because of a call to **fclose** or because the program terminates, all the buffers associated with it are automatically flushed.

Return Value: A zero value indicates success. If an error occurs, EOF is returned and the error indicator is set (see feof).

fgetc() function

The **fgetc()** function prototype is as follows.

```
int fgetc(FILE *fp);
```

This function returns the character currently pointed by the internal file position indicator of the specified fp. The internal file position indicator is then advanced by one character to point to the next character.

Return value: The character read is returned as an int value. If the EOF is reached or a reading error happens, the function returns EOF and the corresponding error or eof indicator is set. You can use either ferror or feof to determine whether an error happened or the EOF was reached.

Example:
Write the program reads an existing file called myfile.txt character by character and uses the n variable to count how many dollar characters ($) does the file contain.

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  int c;
  int n = 0;
  fp=fopen ("myfile.txt","r");
  if (fp==NULL) printf("Error opening file");
  else
  {
    do {
      c = fgetc (fp);
      if (c == '$') n++;
    } while (c != EOF);
    fclose (fp);
    printf ("File contains %d$.\n",n);
  }
  return 0;
}
```

Example:
Write the program opens the file called myfile.txt, and counts the number of characters that it contains by reading all of them one by one. Finally the total amount of bytes is printed out.

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  long n = 0;
  fp = fopen ("myfile.txt","rb");
  if (fp==NULL) printf ("Error opening file");
  else
  {
    while (!feof(fp)) {
      fgetc (fp);
      n++;
      }
    fclose (fp);
    printf ("Total number of bytes: %d\n",n);
  }
  return 0;
}
```

Example:
Opens a file called input.txt which has some random text (less than 200 characters), stores each character in an array, then spits them back out into another file called "output.txt" in reverse order:

```
#include <stdio.h>
int main() {
  char c;         /* declare a char variable */
  char name[200]; /* Initialize array of total
                     200 for characters */
  FILE *f_input, *f_output; /* declare FILE
pointers  */
  int counter = 0; /* Initialize variable for
counter to zero */
```

```
  f_input = fopen("input.txt", "r");
  /* open a text file for reading */

  if(f_input==NULL) {
    printf("Error: can't open file.\n");
    return 1;
  }
  else {
    while(1) { /* loop continuously */
      c = fgetc(f_input); /* fetch the next
character */
      if(c==EOF) {
        /* if end of file reached, break out of
loop */
        break;
      }
      else if (counter<200) { /* else put
character into array */
        name[counter] = c;
        counter++; /* increment the counter */
      }
      else {
        break;
      }
    }

    fclose(f_input); /* close input file */

    f_output = fopen("output.txt", "w");
    /* create a text file for writing */

    if(f_output==NULL) {
      printf("Error: can't create file.\n");
      return 1;
    }
    else {
      counter--; /* we went one too step far */
      while(counter >= 0) { /* loop while
counter's above zero */
        fputc(name[counter], f_output);
        /* write character into output file */
        counter--; /* decrease counter */
      }

      fclose(f_output); /* close output file */
      printf("All done!\n");
      return 0;
    }
  }
}
```

Reading one character at a time can be a little inefficient, so we can use **fgets** to read one line at a time. The **fgets()** function prototype is as follows.

```
char *fgets(char *str, int num, FILE *fp);
```

The **fgets()** function reads characters from the file associated with fp into a string pointed to by str until num-1 characters have been read, a newline character is encountered, or the end of the file is reached. The string is null-terminated and the newline character is retained.

Return value: the function returns str if successful and a null pointer if an error occurs.

You can't use an !=EOF check here, as we're not reading one character at a time (but you can use feof).

Example:

Create a file called myfile.txt in Notepad, include 3 lines and put tabs in the last line.

```
111 222 333
444 555 666
777     888     999

#include <stdio.h>
int main()
{
  char c[10];  /* declare a char array */
  FILE *file;  /* declare a FILE pointer  */

  file = fopen("myfile.txt", "r");
  /* open a text file for reading */

  if(file==NULL)
  {
    printf("Error: can't open file.\n");
    /* fclose(file); DON'T PASS A NULL POINTER TO
fclose !! */
    return 1;
  }
  else
  {
    printf("File opened successfully.
Contents:\n\n");

    while(fgets(c, 10, file)!=NULL) {
      /* keep looping until NULL pointer... */
      printf("String: %s", c);
      /* print the file one line at a time  */
    }

    printf("\n\nNow closing file...\n");
    fclose(file);
    return 0;
  }
}
```

Output:

```
File opened successfully. Contents:

String: 111 222 3String: 33
String: 444 555 6String: 66
String: 777     888     9String: 99

Now closing file...
```

The main area of focus is the while loop - notice how I performed the check for the return of a NULL pointer. Remember that passing in char * variable, c as the first argument assigns the line read into c, which is printed off by printf. We specified a maximum number of characters to be 10 - we knew the number of characters per line in our text file is more than this, but we wanted to show that fgets reads 10 characters at a time in this case.

Notice how fgets returns when the newline character is reached - this would explain why 444 and 777 follow the word "String". Also, the tab character, \t, is treated as one character.

Other function:

fseek() function

```
int fseek (FILE *fp, long int offset, int origin);
```

In the above prototype, there are two arguments:

- fp: Pointer to a FILE object that identifies the stream.
- offset: Number of bytes to offset from origin.
- If offset >= 0: set the position indicator toward to the end of file,
- If offset < 0: set the position indicator toward to the beginning of file.
- origin: Position from where offset is added. It is specified by one of the following constants defined in <cstdio>:

Constant	**Value**	**Meaning**
SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current position of the file pointer
SEEK_END	2	End of file

This function sets the position indicator associated with the fp to a new position defined by adding offset to a reference position specified by origin. The End-of-File internal indicator of the file is cleared after a call to this function.

Return Value: If successful, the function returns a zero value. Otherwise, it returns nonzero value.

Example:

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  fp = fopen ( "myfile.txt" , "w" );
  fputs ( "This is an apple." , fp );
  fseek ( fp , -8 , SEEK_END );
  fputs ( " sam" , fp );
  fclose ( fp );
  return 0;
}
```

After this code is successfully executed, the file myfile.txt contains:

```
This is a sample.
```

Example:

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  fp = fopen ( "myfile.txt" , "w" );
  fputs ( "This is an apple." , fp );
  fseek ( fp , 9 , SEEK_SET );
  fputs ( " sam" , fp );
  fclose ( fp );
  return 0;
}
```

After this code is successfully executed, the file myfile.txt contains:

```
This is a sample.
```

rewind() function

```
void rewind (FILE *fp);
```

This function sets the current position indicator associated with fp to the beginning of the file. A call to rewind is equivalent to:

```
fseek (fp, 0, SEEK_SET);
```

except that, unlike fseek, rewind clears the error indicator.

On streams open for update (read+write), a call to rewind allows to switch between reading and writing.

Example:

```
#include <stdio.h>
#include <conio.h>
int main ()
{
  char str [80];
  int n;
  FILE * fp;
  fp = fopen ("myfile.txt","w+");
  for ( n='A' ; n<='Z' ; n++)
    fputc ( n, fp);
  rewind (fp);
  n=0;
  while (!feof(fp))
  {
     str[n]= fgetc(fp);
     n++;
  }
  fclose (fp);
  printf ("I have read: %s \n",str);
  getch();
  return 0;
}
```

A file called myfile.txt is created for reading and writing and filled with the alphabet. The file is then rewinded, read and its content is stored in a buffer, that then is written to the standard output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Example:

```
#include <stdio.h>
int main()
{
  FILE *file;
  char sentence[50];
  int i;

  file = fopen("sentence.txt", "w+");
  /* we create a file for reading and writing */

  if(file==NULL) {
    printf("Error: can't create file.\n");
    return 1;
  }
  else {
    printf("File created successfully.\n");

    printf("Enter a sentence less than 50
characters: ");
    gets(sentence);

    for(i=0 ; sentence[i] ; i++) {
      fputc(sentence[i], file);
    }

    rewind(file); /* reset the file pointer's
position */

    printf("Contents of the file: \n\n");

    while(!feof(file)) {
      printf("%c", fgetc(file));
    }

    printf("\n");
    fclose(file);
    return 0;
  }
}
```

Output depends on what you entered. First of all, we stored the inputted sentence in a char array, since we're writing to a file one character at a time it'd be useful to detect for the null character. Recall that the null character, \0, returns 0, so putting sentence[i] in the condition part of the for loop iterates until the null character is met.

Then we call rewind, which takes the file pointer to the beginning of the file, so we can read from it. In the while loop we print the contents a character at a time, until we reach the end of the file - determined by using the feof function.

Note that it is essential to have the include file stdio.h referenced at the top of your program in order to use any of these functions: fscanf(), fgets(), fgetc(), fflush(), fprintf(), fputs(), fputc(), feof(), fseek() và rewind().

EOF and errors

When a function returns **EOF** (or, occasionally, 0 or NULL, as in the case of fread and fgets respectively), we commonly say that we have reached “end of file” but it turns out that it's also possible that there's been some kind of I/O error. When you want to distinguish between end-of-file and error, you can do so with the feof and ferror functions. **feof(fp)** returns nonzero (that is, “true”) if end-of-file has been reached on the file pointer fp, and **ferror(fp)** returns nonzero if there has been an error.

Notice **feof** returns nonzero if end-of-file has been reached. It does not tell you that the next attempt to read from the stream will reach end-of-file, but rather that the previous attempt (by some other function) already did. (If you know Pascal, you may notice that the end-of-file detection situation in C is therefore quite different from Pascal.) Therefore, you would never write a loop like

```
while(!feof(fp))
   fgets(line, max, fp);
```

Instead, check the return value of the input function directly:

```
while(fgets(line, max, fp) != NULL)
```

With a very few possible exceptions, you don't use feof to **detect** end-of-file; you use feof or ferror to distinguish between end-of-file and error. (You can also use ferror to diagnose error conditions on output files.)

Since the end-of-file and error conditions tend to persist on a stream, it's sometimes necessary to clear (reset) them, which you can do with **clearerr(FILE *fp)**.

What should your program do if it detects an I/O error? Certainly, it cannot continue as usual; usually, it will print an error message. The simplest error messages are of the form

```
fp = fopen(filename, "r");
        if(fp == NULL)
        {
                fprintf(stderr, "can't open
file\n");
                return;
        }
```

or

```
while(fgets(line, max, fp) != NULL)
        {
          ... process input ...
        }

        if(ferror(fp))
           fprintf(stderr, "error reading
input\n");
```

or

```
fprintf(fp, "%d %d %d\n", a, b, c);
 if(ferror(fp))
      fprintf(stderr, "output write error\n");
```

Error messages are much more useful, however, if they include a bit more information, such as the name of the file for which the operation is failing, and if possible **why** it is failing. For example, here is a more polite way to report that a file could not be opened:

```
        #include <stdio.h>      /* for fopen */
        #include <errno.h>      /* for errno */
        #include <string.h>     /* for strerror */

        fp = fopen(filename, "r");
        if(fp == NULL)
        {
                fprintf(stderr, "can't open %s for
reading: %s\n",
                                        filename,
strerror(errno));
```

```
                return;
        }
```

errno is a global variable, declared in <errno.h>, which may contain a numeric code indicating the reason for a recent system-related error such as inability to open a file. The **strerror** function takes an errno code and returns a human-readable string such as “No such file” or “Permission denied”.

An even more useful error message, especially for a “toolkit” program intended to be used in conjunction with other programs, would include in the message text the name of the program reporting the error.

Access to Binary Files

Write data to binary files

```
size_t fwrite(void *buf, size_t sz, size_t n, FILE
*fp)
```

This function writes to file associated with fp, num number of objects, each object size bytes long, from the buffer pointed to by buffer.

Return value: It returns the number of objects written. This value will be less than num only if an output error as occurred.

The **void pointer** is a pointer that can point to any type of data without the use of a TYPE cast (known as a generic pointer). The type size_t is a variable that is able to hold a value equal to the size of the largest object surported by the compiler.

As a simple example, this program write an integer value to a file called MYFILE using its internal, binary representation.

```
#include <stdio.h>  /* header file  */
#include <stdlib.h>
void main(void)
{
 FILE *fp;   /* file pointer */
 int i;

 /* open file for output */
 if ((fp = fopen("myfile", "w"))==NULL){
  printf("Cannot open file \n");
  exit(1);
 }
 i=100;

 if (fwrite(&i, 2, 1, fp) !=1){
  printf("Write error occurred");
  exit(1);
 }
 fclose(fp);
}
```

Read data from binary files

```
size_t fread(void *buf, size_t sz, size_t n, FILE
*fp)
```

fread reads up to n objects, each of size sz, from the file specified by fp, and copies them to the buffer pointed to by buf. It reads them as a stream of bytes, without doing any particular formatting or other interpretation. (However, the default underlying stdio machinery may still translate newline characters unless the stream is open in binary or "b" mode).

Return value: returns the number of items read. It returns 0 (not EOF) at end-of-file.

Example:

```
#include <stdio.h>
int main() {
  FILE *file;
  char c[30]; /* make sure it is large enough to
hold all the data! */
  char *d;
  int n;

  file = fopen("numbers.txt", "r");

  if(file==NULL) {
    printf("Error: can't open file.\n");
    return 1;
  }
  else {
    printf("File opened successfully.\n");

    n = fread(c, 1, 10, file); /* passing a char
array,
                                  reading 10
characters */
    c[n] = '\0';               /* a char array is
only a
                                  string if it has
the
                                  null character
at the end */
    printf("%s\n", c);         /* print out the
string      */
    printf("Characters read: %d\n\n", n);

    fclose(file);           /* to read the file
from the beginning, */
                            /* we need to close and
reopen the file */
```

```
    file = fopen("numbers.txt", "r");

    n = fread(d, 1, 10, file);
            /* passing a char pointer this time -
10 is irrelevant */
    printf("%s\n", d);
    printf("Characters read: %d\n\n", n);

    fclose(file);
    return 0;
  }
}
```

Output:

```
File opened successfully.
111
222
33
Characters read: 10

111
222
333

444
5ive
Characters read: 10
```

The above code: passing a char pointer reads in the entire text file, as demonstrated. Note that the number fread returns in the char pointer case is clearly incorrect. This is because the char pointer (d in the example) must be initialized to point to something first.

An important line is: c[n] = '\0'; Previously, we put 10 instead of n (n is the number of characters read). The problem with this was if the text file contained less than 10 characters, the program would put the null character at a point past the end of the file.

There are several things you could try with this program:

- After reading the memory allocation section, try allocating memory for d using malloc() and freeing it later with free().
- Read 25 characters instead of 10: n = fread(c, 1, 25, file);
- Not bother adding a null character by removing: c[n] = '\0';
- Not bother closing and reopening the file by removing the fclose and fopen after printing the char array.

Binary files have two features that distinguish them from text files: You can jump instantly to any record in the file, which provides random access as in an array; and you can change the contents of a record anywhere in the file at any time. Binary files also usually have faster read and write times than text files, because a binary image of the record is stored directly from memory to disk (or vice versa). In a text file, everything has to be converted back and forth to text, and this takes time.

Besides reading and writing “blocks” of characters, you can use fread and fwrite to do “binary” I/O. For example, if you have an array of int values:

```
int array[N];
```

you could write them all out at once by calling

```
fwrite(array, sizeof(int), N, fp);
```

This would write them all out in a byte-for-byte way, i.e. as a block copy of bytes from memory to the output stream, i.e. **not** as strings of digits as printf %d would. Since some of the bytes within the array of int might have the same value as the \n character, you would want to make sure that you had opened the stream in binary or "wb" mode when calling fopen.

Later, you could try to read the integers in by calling

```
fread(array, sizeof(int), N, fp);
```

Similarly, if you had a variable of some structure type:

```
struct somestruct x;
```

you could write it out all at once by calling

```
fwrite(&x, sizeof(struct somestruct), 1, fp);
```

and read it in by calling

```
fread(&x, sizeof(struct somestruct), 1, fp);
```

Close Files

The funtion for closing a file :

```
int fclose(FILE* fp);
```

This function closes the file associated with the fp and disassociates it. All internal buffers associated with the file are flushed: the content of any unwritten buffer is written and the content of any unread buffer is discarded. Even if the call fails, the fp passed as parameter will no longer be associated with the file.

Return value: If the file is successfully closed, a zero value is returned.

Example:

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  fp = fopen ("myfile.txt","wt");
  fprintf (fp, "fclose example");
  fclose (fp);
  return 0;
}
```